Abstract of thesis entitled

Iso-Recursive Subtyping: New Theory and Extensions

Submitted by

Yaoda Zhou

for the degree of Doctor of Philosophy

at The University of Hong Kong

in January, 2023

The Amber rules are well-known and widely used for subtyping iso-recursive types. They were first briefly and informally introduced in 1985 by Luca Cardelli in a manuscript describing the Amber language. Despite their use over many years, important aspects of the metatheory of the iso-recursive style Amber rules have not been studied in depth or turn out to be quite challenging to formalize.

This dissertation proposes a new theory of iso-recursive subtyping. After revisiting the problem of subtyping iso-recursive types, we introduce a novel declarative specification for Amber-style iso-recursive subtyping. Informally, the specification states that two recursive types are subtypes if all their finite unfoldings are subtypes. With the help of intermediate weakly positive subtyping rules, the Amber rules are shown to be sound and complete with respect to this declarative specification. We then show two variants of sound, complete and decidable algorithmic formulations of subtyping that employ the idea of double unfoldings. Compared to the Amber rules, the double unfolding rules have the advantage of: (1) being modular; (2) not requiring reflexivity to be built-in; (3) leading to an easy proof of transitivity of subtyping; and (4) being easily applicable to subtyping relations that are not antisymmetric. As far as we know, this is the first comprehensive treatment of iso-recursive subtyping dealing with unrestricted recursive types in a theorem prover.

The new formulations not only shed new insights on the theory of subtyping iso-recursive types, but they also enable extensions with more complex features. We show three extensions in this thesis. Firstly, at the type level, we present an extension with record types and intersection types, showing how our new formulations can be applied non-antisymmetric subtyping. Secondly, at the term level, we apply it to a record calculus with merge operators, solving a current open problem for such calculi of how to support recursive types and the binary methods. Finally, we combine iso-recursive types with bounded quantification conservatively, and show that such integration is helpful to encode positive f-bounded polymorphism and subtyping between algebraic datatypes.

An abstract of exactly 323 words

Iso-Recursive Subtyping: New Theory and Extensions

by

Yaoda Zhou
B.S. *Shanghai Jiao Tong University*

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Doctor of Philosophy

at

University of Hong Kong

Acknowledgements

When I graduated from SJTU, I hadn't thought I would pursue a Ph.D. degree one day. I worked for a startup but wasn't happy at that time. Compared with the boring daily job, I realized that I still loved to do some challenging research. I'd like to thank my friends Di Chen, Yifei Xu, Yan Ji, Minghua Deng, and Hanqing Ge, who consoled me and encouraged me to seek a Ph.D. opportunity almost every night during the tough time.

I never know there is a programming language group in Hong Kong, so when I saw the webpage of the HKU PL group I was surprised. As a Cantonese, Hong Kong is such a place that I am very familiar with. Then I contacted my future supervisor, Prof. Bruno C. d. S. Oliveira. The interview with Bruno was the worst performance during my career due to my laptop issue. However, Bruno finally trusted my passion and ability.

Bruno is the best supervisor I have seen. He is always so kind to every student. I felt anxious at the end of my first-year Ph.D. since I had no progress on formalizing the Amber rules. Bruno then spent lots of time helping me think about the iso-recursive subtyping, which is the main topic of this dissertation. It finally became a very wonderful paper and my first publication, and I was more confident on research since then. I appreciate my supervisor for his continuous discussion and academic training with me. Bruno, you teach me how to become a better researcher.

Another important person for me is Litao Zhou, who accompanies me during the period of thesis writing. In my mind, Litao is my cute and smart little brother. We share lots of common points: same family name, both graduated from SJTU, similar hobbies, and some secrets between us. Litao, I am glad that I can be your mentor for guiding and helping you at the early stage of your Ph.D. career. I enjoy every weekly meeting with you, also our daily life. Our relationship is faithful, so as long as you need me, you can trust and rely on me at anywhere anytime.

I also thank my co-author Jinxu Zhao. Without his insightful proof work, we won't have such an amazing TOPLAS publication. Thank my other labmates Xuan Bi, Yanlin Wang, Ningning Xie, Weixin Zhang, Xuejing Huang, Baber Rehman, Yaozhu Sun, Mingqi Xue, Wenjia Ye, Xu Xue, Chen Cui, Jinhao Tan, and Shengyi Jiang, for their valuable sharing in the weekly seminar and discussion.

Some people also helped me during the Ph.D. period. They are Junru Shao, Zheng Guo, Te Zeng, Junqiang Wu, Guangxiong Luo, Cheng Peng, Yufeng Tao, Jian Weng, Yuyang Huang, Yuncong Hu, Haojun Ma, Chen Xia, Shuai Yang, Jiefeng Chen and other people. Sorry I cannot list all of you due to the space reason.

Family has special significance for me. Finally, I want to thank my mother, father, and memorial grandmother, for their persistent support and unconditional love.

Yaoda Zhou

University of Hong Kong

Contents

List of Figures

List of Tables

Part I

Prologue

Chapter 1

Introduction

Modern statically typed programming languages have increasingly sophisticated type systems and subtyping relations. For example, Scala 3 [59], TypeScript [86], Flow [60] or Ceylon [58] include *recursive types* [88], *intersection* and *union* types [14] or *bounded quantification* [37].

Scala also adds features such as *path-dependent types* or *type bounds* [9] to the mix. The combination of such features enables for powerful and flexible type systems and subtyping relations that allow more reusable code and programming abstractions.

Unfortunately, such power comes at a cost. Subtyping relations with those features (or just a subset) are not well understood, or may even lack desirable properties, such *transitivity*, *decidability* or even *type soundness*. There is a well-known history of problems.

The focus of this thesis is on recursive subtyping. That is we study subtyping relations that include recursive types, and rules to relate recursive types via subtyping. We will also study the interactions of recursive subtyping with some other features, including intersection types and bounded quantification.

1.1 The History

1.1.1 Recursive types

Recursive types are used in nearly all languages to define recursive data structures like sequences or trees. They are also used in Object-Oriented Programming every time a method needs an argument or return type of the enclosing class. Recursive types come in two flavours: *equi-recursive types* and *iso-recursive types* [52]. With equi-recursive types a recursive type is equal to its unfolding. With iso-recursive types, a recursive type and its unfolding are only isomorphic. To convert between the (iso-)recursive type and its isomorphic unfolding, explicit folding and unfolding constructs are necessary. A fold expression constructs a recursive type, while an unfold expression opens a recursive type.

The main advantage of equi-recursive types is convenience, as no explicit conversions are necessary. However, a disadvantage is that algorithms for languages with equi-recursive types are quite complex. Furthermore, integrating equi-recursive types in type systems with

```
data List = Nil | Cons Int List
map :: (Int -> Int) -> List -> List
map f Nil          =
    Nil
map f (Cons x xs)  =
    Cons (f x) (map f xs)
```

```
class Shape {
    int area() {...}
    boolean compareArea(Shape s) {
        return s.area() == area();
    }
    Shape clone() {return new Shape();}
}
```

Figure 1.1: Recursive types in Haskell (left) and Java (right).

advanced type features, while retaining desirable properties such as decidable type-checking, can be hard (or even impossible) [112, 65].

Many programming languages adopt an iso-recursive formulation. In practice, the inconvenience of iso-recursive types is mostly eliminated by "hiding" the explicit folding and unfolding in other constructs. For example, in functional languages, such as Haskell [71] or OCaml [79], a flavor of iso-recursive types is provided via datatypes.

Figure 1.1 (left) illustrates a simple recursive type in Haskell. The `List` datatype is recursive, as the `Cons` constructor requires a `List` as the second argument. Functions such as `map`, can then be defined by pattern matching. While there are no explicit folding or unfolding operations in the program, every use of the constructors (`Nil` and `Cons`) triggers folding of the recursive type. Conversely, the patterns on `Nil` and `Cons` trigger unfolding of the recursive type. Similarly, in nominal Object-Oriented (OO) languages such as Java, iso-recursive types can be introduced in class definitions such as the one to the right of Figure 1.1. This class definition requires recursive types because both `compareArea` and `clone` need to refer to the enclosing class. Like the Haskell program above, there are no explicit uses of folding and unfolding. Instead, constructors trigger folding of the recursive type; while method calls (such as `area()`) trigger recursive type unfolding. The relationship between iso-recursive types, algebraic datatypes and pattern matching, and nominal OO class definitions is well-understood in the research literature [113, 115, 100, 83, 118].

1.1.2 Recursive subtyping

For adding recursive types to a language with subtyping, it is desirable to have *recursive subtyping* between recursive types. The first rules for recursive subtyping, due to Cardelli [32], are the well-known Amber rules [32]. Recursive subtyping has been studied in two different forms: *equi-recursive* subtyping [5, 24, 62], and *iso-recursive* subtyping [84, 18].

Interestingly, the theory for algorithmic subtyping of iso-recursive types has received little attention in the past. The Amber rules are well-known and widely used for subtyping iso-recursive types. They were briefly and informally introduced in 1985 by Cardelli in a manuscript describing the Amber language [32]. Later on, Amadio and Cardelli [5] made a comprehensive study of the theory of recursive subtyping for a system with equi-recursive types employing Amber-style rules. One nice result of their study is a declarative model for specifying when two recursive types are in a subtyping relation. In essence, two (equi-)recursive types are subtypes if their infinite unfoldings are subtypes. Amadio and Cardelli's

study remains to the day a standard reference for the theory of equi-recursive subtyping, although newer work simplifies and improves on the original theory [24, 62]. Since then variants of the Amber rules have been employed multiple times in a variety of calculi and languages, but often in an iso-recursive setting [1, 56, 18, 114, 44, 83]. Perhaps most prominently the seminal work on "*A Theory of Objects*" by Abadi and Cardelli [1] employs iso-recursive style Amber rules.

The Amber rules are appealing due to their apparent simplicity, but the metatheory for their iso-recursive formulation is not well studied. Unlike an equi-recursive formulation, which has a clear declarative specification, there is no similar declarative specification for an iso-recursive formulation so far. Moreover, there are fundamental differences between equi-recursive and iso-recursive subtyping: while equi-recursive subtyping deals with infinite trees and is naturally understood in a coinductive setting [24, 62], an Amber-style iso-recursive formulation deals with finite trees and ought to be understood in an inductive setting. Furthermore, important properties for algorithmic versions of the iso-recursive Amber rules are lacking or are quite difficult to prove. In particular, there is very little work in the literature regarding proof of transitivity for algorithmic formulations of the Amber rules.

Finally, a fundamental lemma that arises in proofs of type preservation for calculi with iso-recursive subtyping is:

$$\text{If } \mu\alpha.\ A \leq \mu\alpha.\ B \text{ then } [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B$$

We call this lemma the *unfolding lemma*. The unfolding lemma plays a similar role in preservation to the substitution lemma (which is needed for proving preservation of beta-reduction), and is used to prove the case dealing with recursive type unfolding. The proof for the unfolding lemma is non-trivial, but there is also little work on proofs of this lemma for the Amber rules. While there are some interesting alternatives for iso-recursive subtyping [72, 84], Amber-style subtyping strikes a good balance between expressive power and simplicity, and is widely used. Thus understanding Amber-style subtyping further is worthwhile.

1.1.3 Record calculi

Record calculi with a concatenation operator have attracted the attention of researchers due to their ability to give the semantics of object-oriented languages with multiple inheritance [50, 30, 36]. The foundational work by Cook and Palsberg [50], and Cardelli [30] work on the semantics of the Obliq language are prime examples of the usefulness of untyped record calculi with record concatenation to model the semantics of OOP with inheritance.

Unfortunately, typed calculi with record concatenation and subtyping have proven to be quite challenging to model. An important problem, identified by Cardelli and Mitchell [36], is that subtyping can hide static type information that is needed to correctly model (common forms of) record concatenation. Cardelli and Mitchell [36] illustrate the problem with a simple example:

```
let f2 (r:{x:Int}) (s:{y:Bool}) : {x:Int} & {y:Bool} = r,,s
```

```
in f2 ({x=3, y=4}) ({y=true, x=false})
```

Here `f2` is a function that takes two records (`r` and `s`) as arguments, and returns a new record that concatenates the two records (`r ,, s`). For the return type of `f2` we use record type concatenation (here denoted as `R & S`). Because of subtyping it is possible to invoke `f2` with records that have *more* fields than the fields expected by the static types of the arguments of `f2`. For instance, while the static type of the first argument of `f2` is `{x : Int}`, the record that is actually provided at the application (`{x = 3, y = 4}`) also has an extra field `y`.

The program above is fine from a typing point of view, but what should the program evaluate to? There are a few common options for the semantics of record concatenation. Record concatenation can be *symmetric*, only allowing the concatenation of records without conflicts; or it can be *asymmetric*, implementing an overriding semantics where, in case of conflicts, fields on the left (or the right) record are given preference. Choosing a naive form of asymmetric concatenation does not work. For instance, with left-biased concatenation, the example above would evaluate to `{x = 3, y = 4}`, which has the *wrong type*! Therefore, Cardelli and Mitchell [36] state that:

> *we should now feel compelled to define R & S only when R and S are* **disjoint***: that is when any field present in an element of R is absent from every element of S, and vice versa.*

hinting for an approach with symmetric concatenation, based on disjointness. But a naive symmetric concatenation operation would result in a record `{x = 3, y = 4, y = true, x = false}` with conflicts, which should not be allowed! Thus, such a naive form of symmetric concatenation does not work either.

1.1.4 Bounded quantification

Bounded quantification was introduced by Cardelli and Wegner [37] in the Fun language, and has been widely studied [53, 35, 98]. Bounded quantification addresses the interaction between parametric polymorphism and subtyping, allowing polymorphic variables to have subtyping bounds.

From the mid-80s and throughout the 90s there was a lot of work on establishing the type-theoretic foundations for OOP. Both recursive subtyping, as well as bounded quantification played a major part on this effort. The two features were perceived to be quite fundamental to model objects. At that time the key ideas around $F_{\leq}$ [53, 37, 35], which is a polymorphic calculus with bounded quantification (but no recursive types) were reasonably well understood due to the early work on the Fun language by Cardelli and Wegner. Therefore $F_{\leq}$-like calculi were being used in foundational work on OOP. Some landmark papers on the foundations of OOP, which established important results such as the distinction between inheritance and subtyping [49], *f-bounded quantification* [29], or encodings of objects [49, 3, 28], all essentially assumed some $F_{\leq}$ variant with recursive types. Typically, recursive subtyping was supported via the Amber rules. However, $F_{\leq}$ itself, as well as the extensions of $F_{\leq}$ with recursive types, had still not been developed and formally studied when many of those works were published.

Unfortunately, the combination of those different features turns out to have unsatisfactory properties later. One of the first, and the most well-known, problematic instance of a subtyping relation is that of (full) $F_{\leq}$, which is undecidable [98]. In this case the language integrates parametric polymorphism with subtyping, leading to the seemingly natural notion of bounded quantification. While in isolation parametric polymorphism and simple subtyping have various desirable properties, the combination and addition of bounded quantification leads to undecidability of subtyping. Even if we take decidable variants of $F_{\leq}$ [37], adding other natural features, such as f-bounded quantification, can once again introduce problems [81, 70].

On the other hand, after the first formalization of $F_{\leq}$ [53], Ghelli [65] questioned the state-of-affairs of bounded quantification with recursive types, which implicitly assumed that the extension of $F_{\leq}$ with recursive types was straightforward. He conducted the first formal study for such an extension, and showed a wide range of negative results. Most importantly, he showed that equi-recursive types are not conservative over full $F_{\leq}$. In other words, adding equi-recursive types to full $F_{\leq}$ changes the expressive power of the subtyping relation, *even when the types being compared do not involve any recursive types.*

The simple addition of equi-recursive types allows well-formed, but invalid subtyping statements in $F_{\leq}$ to be valid in an extension with recursive types. Ghelli also shows that applying equi-recursive types to full $F_{\leq}$ invalidates transitivity elimination: we cannot drop the transitivity rule without losing expressive power. In addition, while subtyping in full $F_{\leq}$ is undecidable [98], the change in expressive power, reopens questions about the decidability or undecidability of the system.

Even if we choose the weaker form of bounded quantification present in Fun language and kernel $F_{\leq}$, the natural extension of Amadio and Cardelli [5]'s algorithm to kernel $F_{\leq}$ is incomplete [46]. Nevertheless, instead of Amadio and Cardelli's *meet 2 times* rules, Colazzo and Ghelli [46] gave an alternative *meet 3 times* algorithm, accompanied by a very challenging correctness proof, showing that the subtyping relation is transitive and complete, but did not prove conservativity. Based on an earlier draft from Colazzo and Ghelli [46], Jeffrey [80] extended the system and proved it correct and complete. By transferring the polar bisimulations [109] technique from concurrency theory, Jeffrey's system is more general than Colazzo and Ghelli's, but it is only partially decidable. It is decidable for kernel $F_{\leq}^{equi}$, but for full $F_{\leq}^{equi}$, only when the algorithm terminates it returns the correct answer, but it may not terminate. Furthermore, although more powerful, Jeffrey's full $F_{\leq}^{equi}$ is not conservative over $F_{\leq}$ either.

Perhaps motivated by the technical challenges and negative results posed by equi-recursive types, some researchers set their sights on iso-recursive types. In their work on object encodings, Abadi et al. [3] proposed the $F_{<:\mu}$ calculus, which supports bounded universal types, bounded existential types and iso-recursive types via the Amber rules. However, reflexivity and transitivity are built-in, so the system is not algorithmic. Furthermore, while they presented the typing, subtyping and reduction rules, they have not proved any properties, including type-soundness or the conservativity over full $F_{\leq}$. One potential reason for the absence of technical results is that the iso-recursive Amber rules are hard to work with formally, as

we mentioned in Chapter 1.1.2, which is difficult to prove results such as transitivity, or define sound and complete algorithmic formulations.

More recently, there has been a flurry of work on the foundations for Scala via the DOT calculus [7, 9, 108, 6, 117, 66, 74, 85]. Early work [9, 108] discovered that transitivity elimination for DOT's declarative subtyping, to obtain an algorithmic formulation, was very challenging. That work also lead to the discovery that the algorithms used in Scala implementations at the time broke transitivity and even type soundness. While Scala 3 has on the meantime fixed known type soundness problems based on the work on DOT, as far as we know it still lacks transitivity of subtyping. Some of the latest work on DOT, has proved that subtyping is undecidable [74], and known decidable fragments lack transitivity [74, 85].

1.2 Contributions

Given the history of problematic interactions of features in subtyping relations, it is perhaps not too surprising that the theory of complex subtyping relations such as the one in DOT or other languages is very challenging. Even for subsets of the features in those subtyping relations, it is not known whether it is possible to obtain well-behaved subtyping relations. Therefore, we believe it is useful to take a step back and study such features and interactions among them more closely. Concretely, in this thesis, we study the following questions:

1.2.1 A new theory for iso-recursive subtyping

We revisit the problem of subtyping iso-recursive types. We start by introducing a novel declarative specification for Amber-style iso-recursive subtyping. Informally, the specification states that two recursive types have subtyping relation if all their finite unfoldings have subtyping relation. More formally, the subtyping rule for recursive types is:

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A]^n\ A \leq [\alpha \mapsto B]^n\ B \qquad \forall n = 1 \cdots \infty}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}\ \text{S-Rec}$$

Here the notation $[\alpha \mapsto A]^n$ denotes the n-times finite unfolding of a type. The n times unfolding applies $n-1$ substitutions to the type A (the recursive type body), and the rule checks that all n-times unfoldings are subtypes. Such a declarative formulation plays a similar role to Amadio and Cardelli's declarative specification for equi-recursive types. Because the specification is defined with respect to the finite unfoldings, this naturally leads to an inductive treatment of the theory. For example, the proof of transitivity of subtyping is fairly straightforward, with the more significant challenge being the unfolding lemma. With all the metatheory in place, proving subject-reduction for a typed lambda calculus with recursive types is a routine exercise. Moreover, the Amber rules are shown to be equivalent (in terms of expressive power) to this declarative specification.

We also show alternative algorithmic formulations based on the idea of *double unfoldings*. We discuss two variants of rules for subtyping recursive types. The first variant, which we call the double unfolding rule, checks both 1-time and 2-times finite unfoldings. The second

variant can be seen as an optimization that checks only 2-times finite unfoldings, by tracking the names of the recursive types to avoid the 1-time finite unfolding check. We call the second variant *nominal unfolding*. Both rules accept all valid subtyping statements that the Amber rules accept, but they have important advantages. In particular the rules with double unfoldings:

- **Enable modular proofs.** The new subtyping rules for recursive types are modular in the sense that proofs for properties such as transitivity or reflexivity only need to account for the new recursive case. All the other cases remain essentially the same as in a subtyping relation without recursive types. Key to this form of modularity is the use standard environments, which are just a collection of type variables.
- **Have easy proofs of transitivity of subtyping.** A particular consequence of the previous point is an easy proof for transitivity, which has been a stumbling block in the past for the iso-recursive Amber rules. The Amber rules have a pervasive impact in the subtyping relation, which is the root cause of the difficulties in doing proofs such as transitivity. To our knowledge the only transitivity proof for the Amber rules is due to Bengtson et al. [18], and the proof is quite intricate, relying on a complex inductive argument.
- **Do not require built-in reflexivity.** An additional benefit is that reflexivity does not have to be built in, but it can be derived instead. In the Amber rules built-in reflexivity is necessary to deal with contravariant occurrences of recursive type variables.
- **Are applicable to non-antisymmetric subtyping relations.** Built-in reflexivity can be problematic in some settings, including calculi with record subtyping or intersection/union types. Such calculi can have "isomorphic" subtyping where two syntactically different types A and B can be subtypes of each other. In other words the subtyping relation is not antisymmetric. Avoiding built-in reflexivity makes the rules easier to apply in such settings. As we show, the double unfolding rules can deal with record types easily.

The focus of our work is on iso-recursive subtyping rules that enable easy metatheory, and improving the understanding of Amber-style iso-recursive subtyping. Therefore our work will be useful to those interested on the theory of recursive types, as well as for formalizations of calculi using iso-recursive subtyping. Formalizations can benefit from our work to easily develop calculi with recursive types and prove important properties, such as transitivity, decidability and type soundness. While the rules based on double unfolding rules are algorithmic and therefore can be used in implementations, our focus is not on efficient algorithms. For implementations, the use of the Amber rules may still be preferable if efficiency is an important concern. Moreover, there are alternatives to the Amber rules, such as the complete rules by Ligatti et al. [84], which may be preferable for extra expressive power in the subtyping relation, as well as efficient algorithms.

To validate all our results we have mechanically formalized all our results in the Coq theorem prover. As far as we know this is the first comprehensive treatment of iso-recursive

subtyping dealing with unrestricted recursive types in a theorem prover.

1.2.2 Intersection types, recursive types and record concatenation

We apply nominal unfoldings to a calculus with a *merge operator* [106, 57] and *disjoint intersection types* [89, 4, 20, 21]. Such calculi *generalize* record calculi with subtyping and a record concatenation operation, which are useful to model the semantics of OO languages with forms of multiple inheritance [31, 50, 36, 30]. Moreover, calculi with disjoint intersection types have been used in recent years to model highly modular and compositional styles of programming [119]. Languages built on top of calculi with disjoint intersection types, can naturally solve the Expression Problem [116], and model very expressive and dynamic forms of inheritance, while retaining static type-safety [19, 119]. Unfortunately, no calculi with disjoint intersection types has support for iso-recursive types, limiting their applicability. We present an iso-recursive extension of the λ_i calculus [89, 77], called λ_i^μ, therefore addressing this open problem.

With λ_i^μ we can use a standard encoding of objects using recursive types [28, 31, 50, 49] in λ_i^μ to model objects with recursive types. For instance, we can define an interface for arithmetic expressions Exp using a recursive type:

$$\mathsf{Exp} := \mu\mathsf{Exp}.\ \{\mathsf{eval} : \mathsf{nat},\ \mathsf{dbl} : \mathsf{Exp},\ \mathsf{eq} : \mathsf{Exp} \to \mathsf{bool}\}$$

In Exp there are 3 methods: an evaluation method that returns the value of evaluating the expression; a dbl method that doubles all the natural numbers in (the AST of) an expression; and an equality method that compares the expression with another expression. In λ_i it is only possible to express the type of eval. However, in λ_i^μ we can also express dbl and eq. Importantly, in λ_i^μ the record type $\{\mathsf{eval} : \mathsf{nat},\ \mathsf{dbl} : \mathsf{Exp},\ \mathsf{eq} : \mathsf{Exp} \to \mathsf{bool}\}$ is syntactic sugar for *intersections of single field records* [106, 57]. In other words, to define the type Exp we need both intersection types and recursive types.

To implement Exp we first need a few auxiliary functions ($\mathsf{eval}' : \mathsf{Exp} \to \mathsf{nat}$, $\mathsf{dbl}' : \mathsf{Exp} \to \mathsf{Exp}$ and $\mathsf{eq}' : \mathsf{Exp} \to \mathsf{Exp} \to \mathsf{bool}$) that unfold the recursive type[1]. Then we define two recursive functions $\mathsf{lit} : \mathsf{nat} \to \mathsf{Exp}$ and $\mathsf{add} : \mathsf{Exp} \to \mathsf{Exp} \to \mathsf{Exp}$:

$$\begin{array}{lcl}
\mathsf{eval}'\ \mathsf{e} & = & (\mathsf{unfold}\ [\mathsf{Exp}]\ \mathsf{e}).\mathsf{eval} \\
\mathsf{dbl}'\ \mathsf{e} & = & (\mathsf{unfold}\ [\mathsf{Exp}]\ \mathsf{e}).\mathsf{dbl} \\
\mathsf{eq}'\ \mathsf{e}_1\ \mathsf{e}_2 & = & (\mathsf{unfold}\ [\mathsf{Exp}]\ \mathsf{e}_1).\mathsf{eq}\ \mathsf{e}_2 \\
\mathsf{lit}\ \mathsf{n} & = & \mathsf{fold}\ [\mathsf{Exp}]\{\mathsf{eval} = \mathsf{n},\ \mathsf{dbl} = \mathsf{lit}(\mathsf{n} * 2), \\
 & & \qquad \mathsf{eq} = \breve{}\mathsf{e}'.\ (\mathsf{eval}'\ \mathsf{e}' == \mathsf{n}) : \mathsf{Exp} \to \mathsf{bool}\} \\
\mathsf{add}\ \mathsf{e}_1\ \mathsf{e}_2 & = & \mathsf{fold}\ [\mathsf{Exp}]\{\mathsf{eval} = \mathsf{eval}'\ \mathsf{e}_1 + \mathsf{eval}'\ \mathsf{e}_2,\ \mathsf{dbl} = \mathsf{add}\ (\mathsf{dbl}'\ \mathsf{e}_1)\ (\mathsf{dbl}'\ \mathsf{e}_2), \\
 & & \qquad \mathsf{eq} = \breve{}\mathsf{e}'.\ (\mathsf{eval}'\ \mathsf{e}' == \mathsf{eval}'\ \mathsf{e}_1 + \mathsf{eval}'\ \mathsf{e}_2) : \mathsf{Exp} \to \mathsf{bool}\}
\end{array}$$

In this example the functions lit and add act as encodings of classes or traits. The function lit is basic: it stores the literal, a double function and equality functions. In add, operations such

[1] We assume the presence of recursive functions, and that records are lazy in the example.

as eval' have to be called for subexpressions. To check if $2 * 7 = 2 * (3 + 4)$, we can define $\mathsf{e}_1 : \mathsf{Exp} = \mathsf{lit}\ 7$ and $\mathsf{e}_2 : \mathsf{Exp} = \mathsf{add}\ (\mathsf{lit}\ 3)\ (\mathsf{lit}\ 4)$. Then, we check if $\mathsf{eq}'\ (\mathsf{dbl}'\ \mathsf{e}_1)\ (\mathsf{dbl}'\ \mathsf{e}_2)$ is satisfied.

In brief, with λ_i^{μ}, we can model certain forms of *structurally typed* object-oriented programming and Compositional Programming, with an expressive form of inheritance and in the presence of recursive types and *binary methods* [25].

1.2.3 Bounded quantification and recursive subtyping

We present an extension of kernel $F_{\leq}$, called $F_{\leq}^{\mu}$, with iso-recursive types. In $F_{\leq}^{\mu}$ we add iso-recursive subtyping using the nominal unfolding rules. With the nominal unfolding rules, proving transitivity and other properties is easy, also enabling developing algorithmic formulations of subtyping instead. Furthermore, a nice property of the nominal unfolding rules is that they are modular, allowing an existing calculus to be extended with recursive types without major impacts on existing definitions and proofs. In other words they allow reusing most existing metatheory and definitions that existed before the addition of iso-recursive types. Our work shows that the nominal unfolding rules can be integrated modularly into $F_{\leq}$, while retaining desirable properties. In particular, we prove, for the first time, the conservativity of an extension of $F_{\leq}$ with recursive types over the original $F_{\leq}$.

We also add two smaller extensions to $F_{\leq}$. The first one is a generalization of the kernel $F_{\leq}$ rule for bounded quantification that accepts *equivalent* rather than *equal* bounds. The second extension is the use of so-called *structural* folding/unfolding rules, inspired by the structural unfolding rule proposed by Abadi, Cardelli, and Viswanathan [3].

$$\textsc{typing-sunfold}\quad \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq \mu\alpha.\ B}{\Gamma \vdash \mathsf{unfold}\ [A]\ e : [\alpha \mapsto A]\ B} \qquad \textsc{typing-sfold}\quad \frac{\Gamma \vdash e : [\alpha \mapsto B]\ A \qquad \Gamma \vdash \mu\alpha.\ A \leq B}{\Gamma \vdash \mathsf{fold}\ [B]\ e : B}$$

The structural rules add expressive power to the more conventional folding/unfolding rules in the literature, and they enable additional applications. The structural unfolding rule was presented by Abadi et al. [3] for supporting *structural update* in the object calculus that was being encoded into $F_{\leq}$ with iso-recursive types. In their work, the structural unfolding rule is presented with an informal explanation. We provide structural rules for both expressions, together with the formalization of the type soundness for both rules. We illustrate how the structural rules play an important role to model encodings of objects, as well as encodings of algebraic datatypes with subtyping.

We present several results, including: *type soundness*; *transitivity* and *decidability* of subtyping; the *conservativity* of $F_{\leq}^{\mu}$ over $F_{\leq}$; and a sound and complete algorithmic formulation of $F_{\leq}^{\mu}$. Moreover, we also present an extension of $F_{\leq}^{\mu}$, called $F_{\leq\geq}^{\mu}$, which has a bottom type and *lower bounded quantification* in addition to the conventional (upper) bounded quantification of $F_{\leq}$. As we show, lower bounded quantification is particularly interesting to model the subtyping of algebraic datatypes.

1.3 Outline

In summary, the structure/contributions of this thesis are:

Chapter 3 **A declarative specification for iso-recursive subtyping:** We propose a new specification for the iso-recursive subtyping, called *finite unfolding rule*, which is showed to be reflexive and transitive. The unfolding lemma is proven via finite approximation. We also prove the type soundness for the simply typed lambda calculus with iso-recursive subtyping, including preservation and progress theorems.

Chapter 4 **Algorithmic subtyping based on double unfoldings:** We propose two algorithmic variants of iso-recursive subtyping, called *double unfolding rule* and *nominal unfolding rule*. Both two rules are proved to be reflexive, transitive, decidable, and equivalent with respect to the finite unfolding rule. The unfolding lemma can be proved directly on the algorithmic variants.

Chapter 5 **Subtyping with a Weakly Positive Restriction:** We propose another variant based on the weakly positive restriction, called *weakly positive unfolding rule*. With the help of weakly positive subtyping, we prove that our specification is equivalent to the Amber rules via soundness and completeness theorems.

Chapter 6 **Iso-recursive subtyping with non-antisymmetric subtyping:** We show that our new formulations of iso-recursive subtyping can apply to non-antisymmetric subtyping relations, such as record types and intersection types. We discuss the spurious subtyping problem on intersection types and prove that our nominal unfolding rule can avoid it.

Chapter 7 **Iso-recursive subtyping with record concatenation:** We show that our new formulations of iso-recursive subtyping can apply to the calculus with record concatenation via extending an existing calculus of disjoint interaction types and merge operator, called λ_i^{μ}. The desirable properties, such as transitivity and decidability of subtyping, type soundness and the unfolding lemma hold. Our λ_i^{μ} calculus illustrates one application of a subtyping relation with iso-recursive subtyping and intersection types, and addresses an open problem in the previous line of work on disjoint intersection types.

Chapter 8 **Iso-recursive subtyping with bounded quantification:** We show that our new formulations of iso-recursive subtyping can apply to the calculus with bounded quantification. Our $F_{\leq}^{\mu}$ calculus illustrates how to integrate iso-recursive types and kernel $F_{\leq}$. We obtain a transitive and decidable subtyping relation, while the full calculus is shown to be conservative over $F_{\leq}$ and is proven to be type-sound. $F_{\leq\geq}^{\mu}$ is an extension of $F_{\leq}^{\mu}$ with lower bounded quantification and bottom types. Both $F_{\leq}^{\mu}$ and $F_{\leq\geq}^{\mu}$ could serve as the theoretic foundation for object encodings and encodings of algebraic datatypes with subtyping.

Furthermore, in Chapter 2, some necessary background for keeping the thesis as self-contained as possible, including iso-recursive subtyping and object encodings, are presented. We also discuss the related work on iso-recursive subtyping and object encodings in Chapter 9.

We have formalized all the calculi and proofs in this thesis in Coq. All the results are formalized in the Coq theorem prover and can be found at:

`https://github.com/juda/dissertation-artifacts`

This thesis is largely based on the drafts and publications by the author [121, 122, 120, 82], as indicated below.

Chapter 3, 4 and 5 Yaoda Zhou, Bruno C. d. S. Oliveira and Jinxu Zhao. 2020. "Revisiting Iso-Recursive Subtyping". In *Proc. of the 35th ACM International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2020)*.

Yaoda Zhou, Jinxu Zhao and Bruno C. d. S. Oliveira. 2022. "Revisiting Iso-Recursive Subtyping". In *ACM Transactions on Programming Languages and Systems (TOPLAS)*.

Chapter 7 Yaoda Zhou, Bruno C. d. S. Oliveira and Andong Fan. 2022. "A Calculus with Recursive Types, Record Concatenation and Subtyping". In *Proc. of the 20th Asian Symposium on Programming Languages and Systems (APLAS 2022)*.

Chapter 8 Litao Zhou*, Yaoda Zhou* and Bruno C. d. S. Oliveira. 2023. "Recursive Subtyping for All". In *Proc. of the 50th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2023)*, conditionally accepted. (* Equal contribution)

Chapter 2

Background

This chapter sets the stage for some basic concepts that will be presented in later chapters. In Chapter 2.1 recursive subtyping is introduced and discussed. In particular, we review the Amber rules [5], which have been widely used for recursive subtyping in the past. In Chapter 2.2 there is some discussion about the encoding of objects and datatypes, including intersection types, merge operators and bounded quantification. There is plenty of their related work to those topics, which is discussed on the Chapter 9.

2.1 Subtyping Recursive Types

Subtyping is a widely-used inclusion relation that compares two types. Many calculi have no types of "infinite" size. In such calculi comparing two types is relatively easy. However, with the existence of recursive types, comparing two types is no longer trivial. A recursive type $\mu\alpha.\ A$ usually contains itself as a subpart, represented by the type variable α. Therefore, a subtyping relation (or another form of comparison) needs to treat these types in a special way.

We choose to use a minimal set of types throughout this work for illustration. A type A, B, C, or D may refer to the primitive nat type, the top type $\top$, a function type $A \to B$, a type variable α or a recursive type $\mu\alpha.\ A$. The subtyping rules for the top type, primitive types and function types are standard:

$$\frac{}{A \leq \top} \qquad \frac{}{\mathsf{nat} \leq \mathsf{nat}} \qquad \frac{B_1 \leq A_1 \qquad A_2 \leq B_2}{A_1 \to A_2 \leq B_1 \to B_2}$$

Before diving into the design of subtyping relations for recursive types, we first look at some examples. We also discuss the role of the *unfolding lemma* in checking whether a subtyping relation between two recursive types is valid or not.

Example 1 Any type should be a subtype of itself, including[1]

- $\mu\alpha.\ \alpha \to \alpha \leq \mu\alpha.\ \alpha \to \alpha$,
- $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \mathsf{nat}$,

[1]We assume that recursive types have lower priority. That is, $\mu\alpha.\ \top \to \alpha$ means $\mu\alpha.\ (\top \to \alpha)$ not $(\mu\alpha.\ \top) \to \alpha$.

- $\mu\alpha.\ \mathsf{nat} \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \alpha$.

An important aspect to pay attention to here is the negative occurrences of recursive type variables, which occur in the first two examples. The combination of contravariance of function types and recursive types is a key cause to some complexity which is necessary when subtyping recursive types, even for the case of equal types. Indeed, this is the key reason why in the Amber rules a reflexivity rule is needed. We will come back to this point in Chapter 2.1.4.

Example 2 A second example is $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \alpha$. This example illustrates *positive* recursive subtyping, since the recursive variables are used only in positive positions, and the two types are not equal. The left type is a function that consumes infinite values of any type, and the right type consumes infinite nat values. Hence, the left type is more general than the right type.

Example 3 The type $\mu\alpha.\ \alpha \to \mathsf{nat}$ is *not* a subtype of $\mu\alpha.\ \alpha \to \top$. This final example serves the purpose of illustrating *negative* recursive subtyping, where recursive type variables occur in negative positions. If we ignore the recursive parts of these types, $A \to \mathsf{nat} \leq A \to \top$ holds for any type A. But that does not imply that $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \top$, because the type variable α on different sides refers to different types. If we unfold both types twice, we get:

$$((\mu\alpha.\alpha \to \mathsf{nat}) \to \mathsf{nat}) \to \mathsf{nat} \quad \text{v.s.} \quad ((\mu\alpha.\alpha \to \top) \to \top) \to \top$$

which should be rejected by the subtyping relation. Because of the contravariance of functions, we need to check not only that $\mathsf{nat} \leq \top$ but also that $\top \leq \mathsf{nat}$ (which does not hold).

The role of the unfolding lemma In Example 3 we argued that subtyping should be rejected without actually defining a rule for subtyping of recursive types. The argument was that in such case subtyping should be rejected because unfolding the recursive type a few times leads to a subtyping relation that is going to be rejected by some other rule not involving recursive types. The unfolding lemma captures the essence of this argument formally:

$$\text{If } \mu\alpha.\ A \leq \mu\alpha.\ B \text{ then } [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B$$

It states that unfolding the types one time in a valid subtyping relation between recursive types always leads to a valid subtyping relation between the unfoldings. This property plays an important role in type soundness, and it essentially guarantees the type preservation of recursive type unfolding.

In the following subsections, we briefly review some possible designs for recursive subtyping.

2.1.1 A Rule That Only Works for Covariant Subtyping

As observed by Amadio and Cardelli [5], a first idea to compare two recursive types is to use the following rules:

$$\frac{\Gamma, \alpha \vdash A \leq B}{\Gamma \vdash \mu\alpha.\, A \leq \mu\alpha.\, B} \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha \leq \alpha}$$

which accept, for example, $\mu\alpha.\, \top \to \alpha \leq \mu\alpha.\, \mathsf{nat} \to \alpha$ and $\mu\alpha.\, \alpha \to \alpha \leq \mu\alpha.\, \alpha \to \alpha$. Unfortunately, these rules are unsound in the presence of negative recursive subtyping and contravariant subtyping for function types. We can easily derive the following invalid relation with those rules:

$$\mu\alpha.\, \alpha \to \mathsf{nat} \leq \mu\alpha.\, \alpha \to \top$$

If we ignore the recursive symbol μ, it is not immediately obvious that the subtyping relation is problematic:

$$\alpha \to \mathsf{nat} \leq \alpha \to \top$$

However, after unfolding the types twice the problem becomes obvious, as shown in Example 3:

$$((\mu\alpha.\, \alpha \to \mathsf{nat}) \to \mathsf{nat}) \to \mathsf{nat} \leq ((\mu\alpha.\, \alpha \to \top) \to \top) \to \top$$

Generally speaking, these rules are sound for positive recursive subtyping. However, contravariant recursive types, where the recursive type variables occur in negative positions, may allow *unsound* subtyping statements, as shown above.

2.1.2 The Positive Restriction Rule

To fix the unsound rule in the presence of contravariant subtyping, we might restrict it with *positivity checks* on the types:

$$\frac{\Gamma, \alpha \vdash A \leq B \qquad \text{non-neg}(\alpha, A) \qquad \text{non-neg}(\alpha, B)}{\Gamma \vdash \mu\alpha.\, A \leq \mu\alpha.\, B}$$

where $\text{non-neg}(\alpha, A)$ is false when α occurs in negative positions of A. This restriction, which was also observed by Amadio and Cardelli [5], solves the unsoundness problem and is employed in some languages and calculi [12]. The logic behind this restriction is that all the subderivations which encounter $\alpha \leq \alpha$ (for some recursive type variable α) are valid. Since such subderivations only occur in positive (or covariant) positions, the left α represents $\mu\alpha.\, A$, and the right α represents $\mu\alpha.\, B$. Since the subtyping is covariant, the statement $\mu\alpha.\, A \leq \mu\alpha.\, B$ is valid, and all substatements $\alpha \leq \alpha$ are valid as well.

The main drawback of this rule is that no negative recursive subtyping is possible. It rejects some valid relations, such as $\mu\alpha.\, \top \to \alpha \leq \mu\alpha.\, \alpha \to \alpha$. Furthermore, at least without some form of reflexivity built-in, it even rejects subtyping of equal types with negative recursive variables, such as $\mu\alpha.\, \alpha \to \alpha \leq \mu\alpha.\, \alpha \to \alpha$.

$\boxed{\Gamma \vdash A \leq B}$ *(Original Amber Rules)*

$$\textsc{OAmber-refl}\ \frac{A = B}{\Gamma \vdash A \leq B} \qquad \textsc{OAmber-trans}\ \frac{\Gamma \vdash A \leq B \qquad \Gamma \vdash B \leq C}{\Gamma \vdash A \leq C} \qquad \textsc{OAmber-assmp}\ \frac{\alpha \leq \beta \in \Gamma}{\Gamma \vdash \alpha \leq \beta} \qquad \textsc{OAmber-top}\ \frac{}{\Gamma \vdash A \leq \top}$$

$$\textsc{OAmber-arrow}\ \frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2} \qquad \textsc{OAmber-rec}\ \frac{\Gamma, \alpha \leq \beta \vdash A \leq B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\beta.\ B}$$

Figure 2.1: The complete Amber subtyping rules by Amadio and Cardelli [5] for *equi-recursive* subtyping.

2.1.3 The Amber Rules

The Amber rules were introduced in the Amber language by Cardelli [32]. Later, Amadio and Cardelli [5] studied the metatheory for a subtyping relation that employs Amber-like rules. These rules are presented in Figure 2.1. The subtyping relation is declarative as the transitivity rule (rule OAmber-trans) is built-in. The rule OAmber-top and rule OAmber-arrow are standard. Rule OAmber-rec is the most prominent one, describing subtyping between two recursive types. The key idea in the Amber rules is to use *distinct* type variables for the two recursive types being compared (α and β). These two type variables are stored in the environment. Later, if a subtyping statement of the form $\alpha \leq \beta$ is found, rule OAmber-assmp is used to check whether that pair is in the environment. The nice thing about rule OAmber-rec and rule OAmber-assmp is that they work very well for positive subtyping. Furthermore, they rule out some bad cases with negative subtyping, such as $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\beta.\ \beta \to \top$. Unfortunately, rule OAmber-rec rules out too many cases with negative subtyping, including statements about equal types, such as $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\beta.\ \beta \to \mathsf{nat}$. To compensate for this, rule OAmber-rec is complemented by a (generalization of the) reflexivity rule (rule OAmber-refl). In the case of Amadio and Cardelli [5]'s original rules, rule OAmber-rec comes with a non-trivial definition of equality $A = B$ (we refer to their paper for details). Such equality allows deriving statements such as $\mu\alpha.\ \mathsf{nat} \to \alpha = \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$ or $\mu\alpha.\ \mathsf{nat} \to \alpha = \mathsf{nat} \to \mu\alpha.\ \mathsf{nat} \to \alpha$, which is used to ensure that recursive types and their unfoldings are equivalent. That is, generally speaking, the following equality holds at the type-level:

$$\mu\alpha.\ A = [\alpha \mapsto \mu\alpha.\ A]\ A$$

In other words, the set of rules defines a subtyping relation for *equi-recursive types*. Amadio and Cardelli [5] did a thorough study of the metatheory of such equi-recursive subtyping, including providing an intuitive specification for recursive subtyping. In essence two recursive types are subtypes if their infinite unfoldings are subtypes.

2.1.4 The iso-recursive Amber rules

Amadio and Cardelli [5]'s set of rules is more powerful than what is normally considered to be the folklore Amber rules for iso-recursive subtyping. Many typical presentations of the Amber rule simply use a variant of syntactic equality[2] in reflexivity, which is less powerful, but it is enough to express iso-recursive subtyping. In what follows we consider the folklore rules, where the equality $(A = B)$ used in rule OAmber-refl is simplified by just considering syntactic equality. The iso-recursive rules can deal correctly with all the examples illustrated so far, accepting the various examples that we have argued should be accepted, and rejecting the other ones. Perhaps a small nitpicking point is the absence of well-formedness constraints in the subtyping rules. By modern day standards, this may look a little suspicious, but then again well-formedness of environments and types is typically standard and straightforward. Unfortunately, as it turns out, a suitable definition of well-formedness is non-trivial for Amber subtyping. We will come back to this issue in Chapter 5. Setting the issue of well-formedness aside for the moment, the Amber rules have some other important issues:

Reflexivity cannot be eliminated The reflexivity rule is essential to the subtyping relation. As we have seen, one cannot even derive $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \mathsf{nat}$ without the reflexivity rule, due to the contravariant positions of the variables. One possible fix is to add another rule that allows variable subtyping in contravariant positions:

$$\frac{\alpha \leq \beta \in \Gamma}{\Gamma \vdash \beta \leq \alpha}$$

However, such rule allows unsound subtypes, for instance, $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \top$. In fact, adding this rule leads to a similar system to that in Chapter 2.1.1.

The reflexivity rule, if present in the subtyping relation, depends on a specific equivalence judgment. Simple systems with antisymmetric subtyping relations might use syntactic equivalence or alpha-equivalence. Yet syntactic or alpha-equivalence might be insufficient for other systems. For example, permutation of fields on record types should be considered as equivalent types, thus we may accept the following subtyping statement:

$$\mu\alpha.\ \{x : \alpha, y : \mathsf{nat}\} \to \mathsf{nat} \leq \mu\alpha.\ \{y : \mathsf{nat}, x : \alpha\} \to \mathsf{nat}$$

However, if the built-in reflexivity employs only alpha-equivalence, such a subtyping statement may be rejected. For instance if record types are modelled as sequences in the abstract syntax (which is quite common [100]), then the two records $\{x : \alpha, y : \mathsf{nat}\}$ and $\{y : \mathsf{nat}, x : \alpha\}$ will be syntactically different. In this case the subtyping relation *is not* antisymmetric. That is both $\{x : \alpha, y : \mathsf{nat}\} \leq \{y : \mathsf{nat}, x : \alpha\}$ and $\{y : \mathsf{nat}, x : \alpha\} \leq \{x : \alpha, y : \mathsf{nat}\}$ are true, but the two types are not equal. Thus, a (strict) reflexivity rule employing syntactic equality is not adequate in such cases. For record types it may be possible to avoid this issue by using a different representation in the abstract syntax. For instance, we could try to model record

[2] More precisely, in a setting where binders and variables are encoded using names, alpha-equivalence is used. In settings where De Bruijn indices are used, it amounts to syntactic equality.

types instead as finite maps from field names to types. Then equality of finite maps could have the expected properties for equality and a standard reflexivity rule could suffice. However, other type system features, such as union ($A \vee B$) and intersection types ($A \wedge B$) [51, 104, 15], would pose similar challenges. In those type systems we wish to have $A \wedge B$ and $B \wedge A$ to be equivalent types, for example. A change of representation of abstract syntax does not seem to help for such features.

The reader may refer to work by Ligatti et al. [84] for a more extended discussion on the complications of having the reflexivity rule built-in. We will also come back to this point in Chapter 6.

Finding an algorithmic formulation: transitivity elimination is non-trivial In the rules that Amadio and Cardelli [5] use, and assuming that equivalence in reflexivity is just alpha-equivalence, simply dropping transitivity (rule OAmber-trans) to obtain an algorithmic formulation loses expressive power. A simple example that illustrates this is:

$$\frac{\alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_3 \vdash \alpha_1 \leq \alpha_2 \qquad \alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_3 \vdash \alpha_2 \leq \alpha_3}{\alpha_1 \leq \alpha_2, \alpha_2 \leq \alpha_3 \vdash \alpha_1 \leq \alpha_3} \text{ DOES NOT HOLD!}$$

Such derivation is valid in a declarative formulation with transitivity, but invalid when transitivity is dropped. Therefore, either the declarative specification must be changed to eliminate "invalid" derivations, or the simply dropping transitivity will not work and some changes in the algorithmic rules are necessary.

Proofs of transitivity and other lemmas are hard A related problem is that proving transitivity of an algorithmic formulation with Amber-style rules is hard. Surprisingly to us, despite the wide use of the Amber rules since 1985 for iso-recursive subtyping, there is very little work that describes transitivity proofs. Many works simply avoid the problem by considering only declarative rules with transitivity built-in [1, 83, 103, 38]. The only proof that we are aware of for transitivity of an algorithmic formulation of the iso-recursive Amber rules is by Bengtson et al. [18]. Some researchers have tried, but failed, to formalize this proof in Coq [12]. They found transitivity is hard to prove syntactically, as it requires a "very complicated inductive argument". Thus, they finally adopt the positive restriction, as we discussed in Chapter 5. We also tried to directly prove some of these properties in Coq with variations of the Amber rules, but none of them works properly.

Non-orthogonality of the Amber rules Finally, the Amber rules interact with other subtyping rules. Besides requiring reflexivity, they require a specific kind of entries in the typing environment, which is different from typical entries in other subtyping relations. This affects other rules, and in particular it affects the proofs for cases that are not related to recursive types. For instance this is a key issue that we encountered when trying to prove transitivity and other properties. Furthermore, it also affects implementations, since adding the Amber rules to an existing implementation of subtyping requires changing existing definitions and some cases

of the subtyping algorithm. In short, the Amber rules are not very modular: their addition has significant impact on existing definitions, rules, implementations and, *most importantly*, proofs.

2.2 Object Encodings

2.2.1 Intersection types and merge operator

Intersection types [104, 51] are widely used in programming languages. For example, they can model key aspects of multiple inheritance [48]. The Merge operator [106] is a term constructor that explicitly introduces a form of intersections. Unfortunately, the merge operator is too powerful to obtain an unambiguous semantics, causing a program to possibly evaluate to different results.

In Chapter 1.1.3, we have discussed the challenges of the combination of record calculi and subtyping. Recent work on calculi with *disjoint intersection types* [89] and a merge operator [106, 57], offers a solution to the Cardelli and Mitchell [36]'s problem for concatenation. The λ_i calculus is one of the calculi in this line of work, featuring a merge operator and disjoint intersection types. In λ_i, all expressions in a merge operator must have disjoint types.

In other words, the λ_i calculus [89] adopts disjointness and restricts subsumption to address the challenges of *symmetric* concatenation/merge. Most importantly, λ_i has a type-directed semantics to ensure proper information hiding and the preservation of the expected modular type invariants. The application of the `f2` function in λ_i results in `{x = 3, y = true }`, which has *no conflicts* and is of the *right type*. Types are used at runtime to ensure that fields hidden by subtyping are dropped from the record. This is enforced, for example, during beta-reduction, which uses the type of the argument to filter any hidden fields/values from records/merges. Thus, before substitution, the first argument of `f2`, for instance, is first filtered using the type `{x : Int}`. The actual record that is substituted in the body of `f2` is `{x = 3}` (and not `{x = 3, y = 4}`).

λ_i and other calculi with disjoint intersection types [89, 4, 20] have been shown to provide flexible forms of *dynamic* multiple inheritance [19, 119]. Moreover, they enable a highly modular and compositional programming style that addresses the Expression Problem [116] naturally. For simplicity, here we only illustrate briefly the ability of such calculi to model *first-class traits* and a very dynamic form of inheritance [19]:

```
addId(super:Trait[Person], idNumber:Int):Trait[Student]=
  trait inherits super => { def id : Int = idNumber }
```

In this code, written in the SEDEL language [19], there are two noteworthy points. Firstly, unlike statically typed mainstream OOP languages, traits (which are similar to OOP classes) are first class. They can be passed as arguments (such as `super`), or returned as a result as above. Secondly, the code uses a highly dynamic form of inheritance. The trait that is inherited (`super`) is a parameter of the function. In contrast, in languages like Java, for `class A extends B`, the class `B` must be statically known. We refer the interested reader to the work by Bi and Oliveira

[19] and Zhang et al. [119] for a much more extensive discussion on the applications of calculi with disjoint intersection types, as well as how to encode source language features, such as first-class traits.

2.2.2 Bounded quantification

Bounded quantification allows types to be abstracted by type variables with a subtyping constraint (or bound). The standard calculus with bounded quantification, $F_{\leq}$ [37, 53, 35], has two common variants when it comes to subtyping universal types. The full $F_{\leq}$ variant [53, 35] compares bounded quantifiers with the following rule:

$$\frac{\Gamma \vdash A_2 \leq A_1 \qquad \Gamma, \alpha \leq A_2 \vdash B \leq C}{\Gamma \vdash \forall(\alpha \leq A_1).\, B \leq \forall(\alpha \leq A_2).\, C} \text{ S-FULLALL}$$

The most significant characteristic of full $F_{\leq}$ is that it allows two bounded quantifiers to be contravariant on their bound types A_1 and A_2 when being compared. However, the rich expressivity of full $F_{\leq}$ results in an undecidable subtyping relation [98], which is undesirable. In addition, as Ghelli [65] demonstrates, the rule S-FULLALL may even prevent conservative extensions of $F_{\leq}$ in the presence of additional features, such as equi-recursive types. Ghelli illustrates this with a simple example:

$$\begin{array}{lcllcl} B & \equiv & \forall\alpha.\ -(\forall(\alpha' \leq \alpha).\ -\alpha) & A' & \equiv & \forall(\beta \leq B).\, \forall(\beta' \leq \beta).\ -\beta \\ A & \equiv & \forall(\beta \leq B).\, \beta & R & \equiv & \forall(\beta \leq B).\, \mu X.\, \forall(\beta' \leq X).\ -X \end{array}$$

where $-A$ stands for $A \to \top$ and $\forall\alpha.\ A$ is the abbreviation of $\forall(\alpha \leq \top).\ A$. In full $F_{\leq}$, $A \leq A'$ does not hold. However, both $A \leq R$ and $R \leq A'$ can be derived in full $F_{\leq}$ when the equi-recursive subtyping is allowed, even the subtyping between recursive types is weaker than strong recursion [5, 65].

There are several ways to restrict bounded quantification to a fragment with decidable subtyping, such as removing top types, or assuming no bounds when comparing type abstraction bodies [41]. Among those the most widely used variant is the kernel $F_{\leq}$ calculus. In kernel $F_{\leq}$ bounded quantifiers can only be subtypes when their bound types are identical [37], which is stated in the rule S-KERNELALL.

$$\frac{\Gamma \vdash A \qquad \Gamma, \alpha \leq A \vdash B \leq C}{\Gamma \vdash \forall(\alpha \leq A).\, B \leq \forall(\alpha \leq A).\, C} \text{ S-KERNELALL}$$

We can further generalize the rule S-KERNELALL to rule S-EQUIVALL that accepts equivalent bounds instead. The use of rule S-EQUIVALL enables more subtyping involving non-antisymmetric subtyping relations, such as records.

$$\frac{\Gamma \vdash A_1 \leq A_2 \qquad \Gamma \vdash A_2 \leq A_1 \qquad \Gamma, \alpha \leq A_2 \vdash B \leq C}{\Gamma \vdash \forall(\alpha \leq A_1).\, B \leq \forall(\alpha \leq A_2).\, C} \text{ S-EQUIVALL}$$

In the Chapter 8, we will focus on kernel $F_{\leq}$, in order to achieve decidable subtyping with iso-recursive subtyping.

Part II

Basic Theory

Chapter 3

A New Specification for Iso-Recursive Subtyping

In Chapter 2.1.4, we describe the iso-recursive Amber rules, which are the most widely-used subtyping rules for iso-recursive types. While the Amber rules are simple, as we have argued, there are important issues with the rules. In particular developing the metatheory for the Amber rules is quite hard. As a first step towards understanding the essence of the Amber rules, in the Chapter 3, we provide a new declarative specification of iso-recursive subtyping in terms of finite unfoldings.

3.1 Overview

The key idea of the new rules is inspired by the rules presented for *covariant subtyping* in Chapter 2.1.1. The logic of the covariant rules is to approximate recursive subtyping using what we call a 1-time *finite unfolding*. We say that the unfolding is finite because we simply use α instead of using the recursive type itself during unfolding. If we apply finite unfoldings to all recursive types, we eventually end up having a comparison of two types representing finite trees. The covariant rules work fine in a setting with covariant subtyping only, but are unsound in a setting that also includes contravariant subtyping. A plausible question is then: can we fix these rules to become sound in the presence of contravariant subtyping?

The answer to this question is yes! Let us have a second look at the unsound counter-example that was presented in Chapter 2.1.1:

$$\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \top$$

As we have argued, this subtyping statement should fail because unfolding the recursive type twice leads to an invalid subtyping statement. However, with the 1-time finite unfolding used by the rules in Chapter 2.1.1, all that is checked is whether $\alpha \vdash \alpha \to \mathsf{nat} \leq \alpha \to \top$ holds. Since such statement does hold, the rule *unsoundly* accepts $\mu\alpha.\ \alpha \to \mathsf{nat} \leq \mu\alpha.\ \alpha \to \top$. The problem is that while the 1-time unfolding works, other n-times unfoldings do not. Therefore, an idea is to check whether other n-times unfoldings work as well to recover soundness.

3.1.1 Declarative subtyping

Our declarative subtyping rules build on the previous observation and only accept the subtyping relation between two recursive types if and only if *all* their n-times finite unfoldings are subtypes for any positive integer n:

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A]^n\ A \leq [\alpha \mapsto B]^n\ B \qquad \forall n = 1 \cdots \infty}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}\ \text{S-Rec}$$

In comparison to the rules showed in Chapter 2.1.1, our subtyping rule S-rec has a stricter condition, by checking the subtyping relation for all n-times finite unfoldings, instead of only the 1-time finite unfolding. Such restriction eliminates the false positives on contravariant recursive types. The definition of n-times finite unfolding used in the rule is as follows:

Definition 1 (n-times finite unfolding)**.**

$$[\alpha \mapsto A]^n\ B := \underbrace{[\alpha \mapsto A][\alpha \mapsto A] \cdots [\alpha \mapsto A]}_{(n-1)\text{ times}}\ B$$

By definition, $[\alpha \mapsto A]^n\ A$ is the n-times finite unfolding of $\mu\alpha.\ A$, but we use a slight generalization (mainly for proofs) to unfold a type B with another type A multiple times. For example, for the recursive type $\mu\alpha.\ \mathsf{nat} \to \alpha$, the 1-time finite unfolding is $\mathsf{nat} \to \alpha$ and the two times finite unfolding is $\mathsf{nat} \to \mathsf{nat} \to \alpha$. Note that the zero-times finite unfolding would be the recursive type itself, according to our terminology. In other words, we execute $(n-1)$ times substitutions (where n corresponds to the arity of the finite unfolding) of the body of the recursive type to itself. For example, $[\alpha \mapsto A]^1\ A = A$, $[\alpha \mapsto A]^2\ A = [\alpha \mapsto A]\ A$, $[\alpha \mapsto A]^3\ A = [\alpha \mapsto A][\alpha \mapsto A]\ A$, etc. The counting scheme for the n-times finite unfolding definition may look at little odd. One may expect the more natural looking definition where the body is unfolded n times instead of $n-1$ times. However, using n times instead of $n-1$ would disagree with our terminology for finite unfoldings of recursive types. For instance, the 1-time unfolding of $\mu\alpha.\ \mathsf{nat} \to \alpha$ is $\mathsf{nat} \to \alpha$, and does *zero* (not one!) substitutions in the body.

In rule S-rec, the number of times that the left and the right types are unfolded is exactly the same. One may wonder if it makes sense to consider cases where we would unfold the recursive types a different number of types on the left and on the right. We believe that such approach would lead to a type unsound rule, and that it is important that the number of finite unfoldings is the same. For instance, consider $\mu\alpha.\ \mathsf{nat} \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \top$. In this case if we choose to unfold the body of the left recursive type $n+1$ times and the body of the right recursive type only n times (for all n) then we would get a valid subtyping statement. However, those two types should not be subtypes since if we apply the unfolding lemma we would obtain: $\mathsf{nat} \to (\mu\alpha.\ \mathsf{nat} \to \alpha) \leq \mathsf{nat} \to \mathsf{nat} \to \top$. The latter is not a valid subtyping statement.

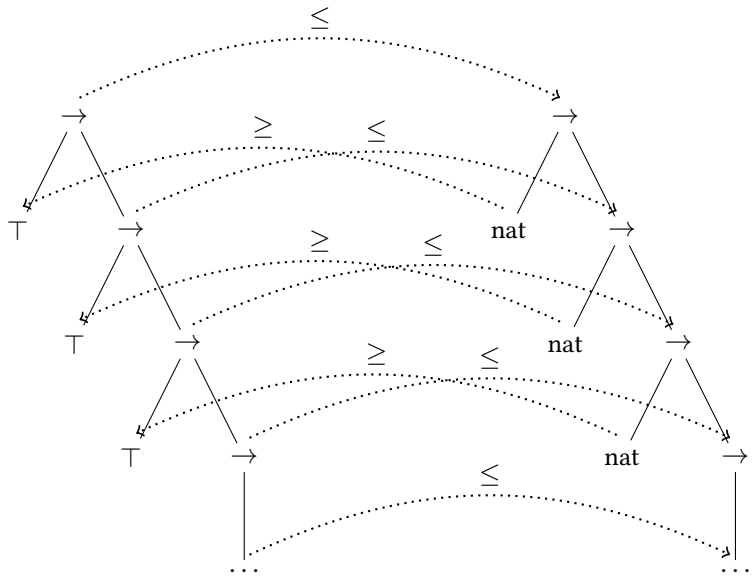


Figure 3.1: Tree model for equi-recursive subtyping.

3.1.2 Contrasting Equi and Iso-Recursive Types

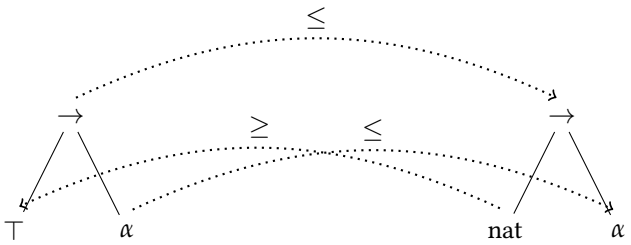
It is useful to contrast the rule S-REC and its formulation in terms of finite unfoldings to Amadio and Cardelli [5]'s specification of equi-recursive subtyping in terms of infinite unfoldings of the recursive types. In Amadio and Cardelli [5]'s work they use the notion of *finite approximation* of a tree, which is closely related to the idea of finite unfoldings. A simplified[1] specification of equi-recursive subtyping in terms of subtyping of infinite trees can be reformulated in terms of finite unfoldings as:

$$\frac{\text{S-EQUI}}{\Gamma \vdash [\alpha \mapsto A]^{\infty}\ A \leq [\alpha \mapsto B]^{\infty}\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}$$

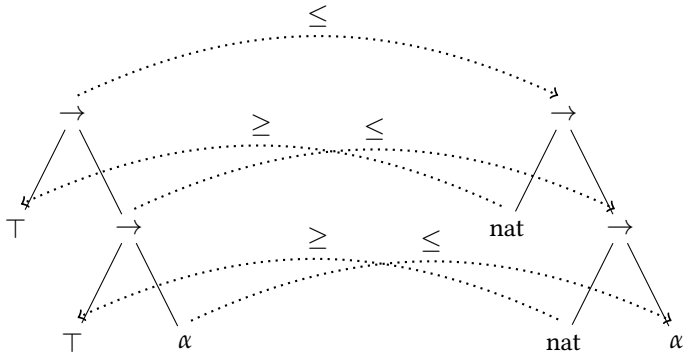
Where the notation $[\alpha \to A]^{\infty}\ A$ denotes applying infinite substitutions to A. In other words, we define the equi-recursive comparison by just one comparison on the limit case, which will potentially compare two infinite trees. With rule S-EQUI subtyping statements such as $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \alpha$ hold, just like with the rule S-REC for iso-recursive subtyping. However, unlike iso-recursive subtyping, subtyping statements such as $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$ also hold, since we unfold both trees to the limit. Figure 3.1 visualizes the tree model equi-recursive subtyping. Note that the figure applies to both $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \alpha$ and $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$, since in both cases the infinite unfoldings of the trees in the subtyping statements are the same.

Instead of a single comparison in the limit case, the rule S-REC for iso-recursive subtyping requires infinitely many comparisons, one for each n-time unfolding. For example, Figure 3.2

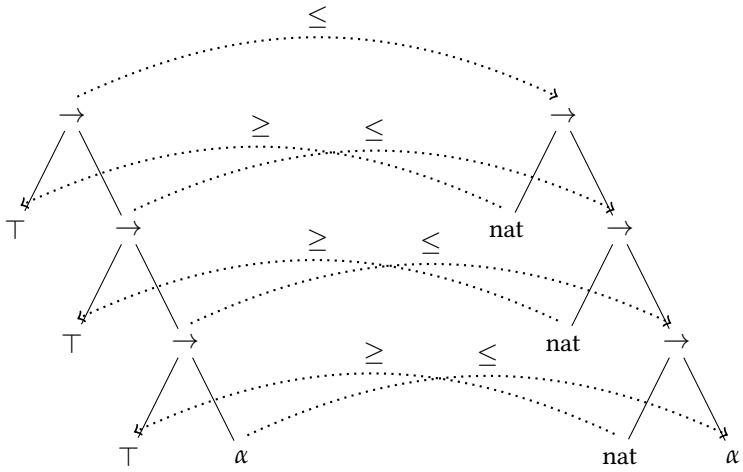
[1] This definition is simplified because the rule S-EQUI compares only two recursive types. In general, in equi-recursive formulations, any two types (recursive or not) can be unfolded and compared. For instance $\mathsf{nat} \to (\mu\alpha.\ \mathsf{nat} \to \alpha) \leq \mu\alpha.\ \mathsf{nat} \to \alpha$ should hold, since the infinite unfoldings of the two types are the same.



(a) Rank-1 tree



(b) Rank-2 tree



(c) Rank-3 tree

Figure 3.2: Tree model for iso-recursive subtyping for the first 3 finite unfoldings for $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ nat \to \alpha$.

visualizes the comparisons for $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \alpha$ in the iso-recursive model. In the figure we show only the first 3 comparisons, which would correspond to the 1-time, 2-times and 3-times finite unfoldings respectively. However, there would be an infinite number of such comparisons for all n-times finite unfoldings. Using rule S-REC the subtyping statement $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$ fails, unlike in the equi-recursive model. It is easy to see why this is the case. Since rule S-REC requires that *all* comparisons are successful, to show that two recursive types are not subtypes it is enough to show that one of the finite comparisons fails. For example, the comparison of 1-time finite unfoldings, which amounts to

$\top \to \alpha \leq \mathsf{nat} \to \mathsf{nat} \to \alpha$, fails. Therefore, we can see that rule S-REC rejects $\mu\alpha.\ \top \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \alpha$.

3.2 Syntax, Well-Formedness and Subtyping

In this section we will introduce a full calculus with declarative subtyping and recursive types. Our calculus is based on the simply typed lambda calculus extended with iso-recursive types and subtyping. This declarative system captures the idea that, with iso-recursive types, two recursive types are subtypes if all their finite unfoldings are subtypes.

3.2.1 Syntax and well-formedness

The calculus that we model is a simply typed lambda calculus with subtyping. The syntax of types and contexts for this calculus is shown below.

Types	A, B, C, D	$::=$	$\mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\ A$
Expressions	e	$::=$	$x \mid \mathsf{i} \mid e_1\ e_2 \mid \lambda x : A.\ e \mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e$
Values	v	$::=$	$\mathsf{i} \mid \lambda x : A.\ e \mid \mathsf{fold}\ [A]\ v$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, \alpha \mid \Gamma, x : A$

Meta-variables A, B, C, D range over types. These types consist of: natural numbers (nat), the top type ($\top$), function types ($A \to B$), type variables (α) and recursive types ($\mu\alpha.\ A$). Expressions, denoted as e, include: term variables (x), natural numbers (i), applications ($e_1\ e_2$), lambda expressions ($\lambda x : A.\ e$). The expression unfold $[A]\ e$ is used to unfold the recursive type of an expression e; while fold $[A]\ e$ is used to fold the recursive type of an expression e. Some expressions are also values: natural numbers (i), lambda expressions ($\lambda x : A.\ e$) as well as fold expressions (fold $[A]\ v$) if their inner expressions are also values. The context is used to store variables with their type and type variables.

The definition of a well-formed environment $\vdash \Gamma$ is standard (Figure 3.3), ensuring that all variables in the environment are distinct. In a well-formed environment, repetition of variables is not allowed and the order of variables are not important. Note that, throughout the paper, we adopt the convention that variables are distinct. For instance, in rule WFT-REC the α introduced in Γ is distinct from other variables in Γ. In our Coq formalization the use of a locally nameless [42] encoding for binders makes such informal conventions precise. The middle of Figure 3.3 also shows the judgement for well-formed types. A type is well-formed if all of its free variables are in the context. The rules of this judgement are mostly standard. The rule WFT-REC states that if the body of a recursive type is well-formed under an extended context then the recursive type is well-formed.

3.2.2 Subtyping

The bottom of Figure 3.3 shows the declarative subtyping judgement. Our subtyping rules are standard with the exception of the new rule for recursive types. Rule S-TOP states that any well-formed type A is a subtype of the $\top$ type. Rule S-VAR is a standard rule for type variables which are introduced when unfolding recursive types: variable α is a subtype of itself. The rule for

$\boxed{\vdash \Gamma}$ *(Well-Formed Environment)*

$$\textsc{wfe-empty}\ \frac{}{\vdash \cdot} \qquad \textsc{wfe-sub}\ \frac{\vdash \Gamma \qquad \alpha \notin \Gamma}{\vdash \Gamma, \alpha} \qquad \textsc{wfe-typ}\ \frac{\vdash \Gamma \qquad x \notin \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, x : A}$$

$\boxed{\Gamma \vdash A}$ *(Well-Formed Type)*

$$\textsc{wft-nat}\ \frac{}{\Gamma \vdash \mathsf{nat}} \qquad \textsc{wft-Top}\ \frac{}{\Gamma \vdash \top} \qquad \textsc{wft-var}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{wft-arrow}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2} \qquad \textsc{wft-rec}\ \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \mu\alpha.\, A}$$

$\boxed{\Gamma \vdash A \leq B}$ *(Declarative Subtyping)*

$$\textsc{s-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nat} \leq \mathsf{nat}} \qquad \textsc{s-top}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash A \leq \top} \qquad \textsc{s-var}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \alpha}{\Gamma \vdash \alpha \leq \alpha} \qquad \textsc{s-arrow}\ \frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2}$$

$$\textsc{s-rec}\ \frac{\Gamma, \alpha \vdash [\alpha \mapsto A]^n\ A \leq [\alpha \mapsto B]^n\ B \quad \forall n = 1 \cdots \infty}{\Gamma \vdash \mu\alpha.\, A \leq \mu\alpha.\, B}$$

Figure 3.3: Well-formedness and subtyping rules.

function types (rule S-ARROW) is standard, but worth mentioning because it is contravariant on input types. As illustrated in Chapter 2.1 (and various previous works), the interaction between recursive types and contravariance has been a key difficulty in the development of subtyping with recursive types. Finally, rule S-REC is the most significant: it tells us that a recursive type $\mu\alpha.\ A$ is a subtype of $\mu\alpha.\ B$, if all their corresponding finite unfoldings are subtypes. Both $[\alpha \mapsto A]^n\ A$ and $[\alpha \mapsto B]^n\ B$ are used to denote n-times finite unfolding, as Definition 1 has illustrated.

3.3 Metatheory of Subtyping

The metatheory of the subtyping relation includes three essential properties: reflexivity, transitivity and the unfolding lemma.

3.3.1 A better induction principle for subtyping properties

The first challenge that we face when looking at the metatheory of subtyping with recursive types is to find adequate induction principles for various proofs. In particular the proofs of reflexivity and transitivity can be non-trivial without a suitable induction principle. A first idea to prove both reflexivity and transitivity is to use induction on well-formed types. However, the problem of using this approach is that there is a mismatch between the well-formedness and subtyping rules for recursive types. The induction hypothesis that we get from rule WFT-REC gives us a statement that works on 1-time finite unfoldings, whereas in the subtyping rule we have a premise expressed in terms of all finite unfoldings.

Fortunately, we can define an alternative variant of well-formedness that gives us a better induction principle. The idea is to replace rule WFT-REC with a rule that expresses that if all finite unfoldings of a recursive type are well-formed then the recursive type is well-formed.

Definition 2. Rule WFT-INF is defined as:

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A]^n\ A \quad \forall n = 1 \cdots \infty}{\Gamma \vdash \mu\alpha.\ A}\ \text{WFT-INF}$$

The two definitions of well-formedness are provably equivalent. In the proofs that follow, when we use induction on well-formed types, we use the variant with the rule WFT-INF.

3.3.2 Reflexivity and transitivity

Next we prove reflexivity and transitivity. First of all, we know that subtyping is regular, i.e. subtyping implies well-formedness of context and types:

Lemma 3. Regularity: If $\Gamma \vdash A \leq B$ then $\vdash \Gamma$ and $\Gamma \vdash A$ and $\Gamma \vdash B$.

Another important property of our subtyping rules is that the order of variables in contexts is irrelevant. That is we can always permute whole portions of the environment:

Lemma 4. If $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4 \vdash A \leq B$ then $\Gamma_1, \Gamma_3, \Gamma_2, \Gamma_4 \vdash A \leq B$.

Thanks to our standard context, the proofs of both reflexivity and transitivity are straightforward using the variant of well-formedness with rule WFT-INF. This contrasts with the Amber rules [32], where reflexivity needs to be built-in and the proof of transitivity is quite complex (and hard to mechanize on a theorem prover) [18, 12].

Theorem 5. Reflexivity.

$$\text{If } \vdash \Gamma \text{ and } \Gamma \vdash A \text{ then } \Gamma \vdash A \leq A.$$

Theorem 6. Transitivity.

$$\text{If } \Gamma \vdash A \leq B \text{ and } \Gamma \vdash B \leq C \text{ then } \Gamma \vdash A \leq C.$$

Proof. From Lemma 3 we know all types and the environment are well-formed. Do induction on $\Gamma \vdash B$ (with the rule WFT-INF).

- Rule WFT-NAT: Do inversion on both two subtyping statements, and we know that A is nat and C is nat or $\top$.
- Rule WFT-TOP: Do inversion on $\Gamma \vdash \top \leq C$, and we know that C is $\top$.
- Rule WFT-VAR: Do inversion on both two subtyping statements, and we know that A is α and C is α or $\top$.
- Rule WFT-ARROW: Assume $B := B_1 \to B_2$.

  - Do inversion on $\Gamma \vdash B_1 \to B_2 \leq C$, we know C is $\top$ or $C := C_1 \to C_2$. The former one is solved immediately. From the latter one, we obtain $\Gamma \vdash C_1 \leq B_1$ and $\Gamma \vdash B_2 \leq C_2$.
  - Do inversion on $\Gamma \vdash A \leq B_1 \to B_2$, we know $A := A_1 \to A_2$ and obtain $\Gamma \vdash B_1 \leq A_1$ and $\Gamma \vdash A_2 \leq B_2$.
  - Now the goal is $\Gamma \vdash A_1 \to A_2 \leq C_1 \to C_2$. Applying the arrow rule, what we need to prove are $\Gamma \vdash C_1 \leq A_1$ and $\Gamma \vdash A_2 \leq C_2$. The two goals can be solved by the induction hypotheses.

- Rule WFT-INF: Assume $B := \mu\alpha.\ B'$.
  - Firstly, it is worthwhile stating the induction hypothesis that we get from rule WFT-INF explicitly: $\forall n\ A\ C,\ \Gamma, \alpha \vdash A \leq [\alpha \mapsto B']^n\ B' \wedge \Gamma, \alpha \vdash [\alpha \mapsto B']^n\ B' \leq C \Rightarrow \Gamma, \alpha \vdash A \leq C$.
  - Do inversion on $\Gamma \vdash \mu\alpha.\ B' \leq C$, we know C is $\top$ or $C := \mu\alpha.\ C'$. The former one is solved immediately. From the latter one, we obtain $\forall n,\ \Gamma, \alpha \vdash [\alpha \mapsto B']^n\ B' \leq [\alpha \mapsto C']^n\ C'$.
  - Do inversion on $\Gamma \vdash A \leq \mu\alpha.\ B'$, we know $A := \mu\alpha.\ A'$ and obtain $\forall n,\ \Gamma, \alpha \vdash [\alpha \mapsto A']^n\ A' \leq [\alpha \mapsto B']^n\ B'$.
  - Now the goal is $\Gamma \vdash \mu\alpha.\ A' \leq \mu\alpha.\ C'$. Applying the rule for recursive types, what we need to prove is $\forall n,\ \Gamma, \alpha \vdash [\alpha \mapsto A']^n\ A' \leq [\alpha \mapsto C']^n\ C'$, which can be solved by the induction hypothesis.

□

Modularity of the proofs Note that in our transitivity proof, all the cases, except for the recursive case, are standard and essentially the same as in a calculus without recursive types. In other words the proof is modular in the sense that existing cases of the proof are not significantly affected by the addition of recursive types. Other proofs, such as reflexivity or weakening, are similarly modular in the same sense. Existing proofs for previous formulations of iso-recursive subtyping [18, 84] and in particular transitivity proofs are non-modular, and require significant changes after the addition of recursive types. We discuss this in more detail in related work (Chapter 9.1).

3.3.3 Unfolding lemma

Next, we turn to the unfolding lemma:

$$\text{If } \Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B \text{ then } \Gamma \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B.$$

which states: if two recursive types are in a subtyping relation, then substituting themselves into their bodies preserves the subtyping relation. This lemma plays a crucial role in the proof

of type preservation as we shall see in Chapter 3.4. However, the lemma cannot be proved directly: we need to prove a generalized lemma first.

Lemma 7. If

1. $\Gamma_1, \alpha, \Gamma_2 \vdash A \leq B$;
2. $\Gamma_1, \Gamma_2 \vdash \mu\alpha_1.\ C$ and $\Gamma_1, \Gamma_2 \vdash \mu\alpha_1.\ D$;
3. α does not occur free in C and D;
4. $\Gamma_1, \alpha_1, \Gamma_2 \vdash [\alpha_1 \mapsto C]^n[\alpha \mapsto \alpha_1]\ A \leq [\alpha_1 \mapsto D]^n[\alpha \mapsto \alpha_1]\ B$ holds for all n,

then $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha_1.\ C]\ A \leq [\alpha \mapsto \mu\alpha_1.\ D]\ B$.

Proof. Induction on $\Gamma_1, \alpha, \Gamma_2 \vdash A \leq B$. Cases rules S-NAT, S-TOP, and S-ARROW are simple.

- Rule S-VAR. Assume that A and B are variable β. If $\beta \neq \alpha$, then the goal is proven directly. Otherwise, the fourth premise is $\Gamma_1, \alpha_1, \Gamma_2 \vdash [\alpha_1 \mapsto C]^n\ \alpha_1 \leq [\alpha_1 \mapsto D]^n\ \alpha_1$, where n is arbitrary. The goal becomes $\Gamma_1, \Gamma_2 \vdash \mu\alpha_1.\ C \leq \mu\alpha_1.\ D$. Then we can apply the rule for recursive types. Note that in the context, the order of variables is unimportant (see Lemma 4) we can permute the context without affecting the correctness. Therefore, the goal is equal to the fourth premise after context permutation and alpha-conversion between α_1 and α, which is possible due to the premise (3). Note also, that premise (3) can be derived from premise (2), but we explicitly show it as a premise due to the role in the proof.
- Rule S-REC. Assume that the shape of A is $\mu\alpha_2.\ A'$ and the shape of B is $\mu\alpha_2.\ B'$.
  - The fourth premise becomes $\forall n,\ \Gamma_1, \alpha_1, \Gamma_2 \vdash [\alpha_1 \mapsto C]^n[\alpha \mapsto \alpha_1]\ \mu\alpha_2.\ A' \leq [\alpha \mapsto D]^n[\alpha \mapsto \alpha_1]\ \mu\alpha_2.\ B'$, which can be rewritten to $\forall n,\ \Gamma_1, \alpha_1, \Gamma_2 \vdash \mu\alpha_2.\ [\alpha \mapsto C]^n[\alpha \mapsto \alpha_1]\ A' \leq \mu\alpha_2.\ [\alpha \mapsto D]^n[\alpha \mapsto \alpha_1]\ B'$.
  - The goal becomes $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha_1.\ C]\ \mu\alpha_2.\ A' \leq [\alpha \mapsto \mu\alpha_1.\ D]\ \mu\alpha_2.\ B'$, which can be rewritten to $\Gamma_1, \Gamma_2 \vdash \mu\alpha_2.\ [\alpha \mapsto \mu\alpha_1.\ C]\ A' \leq \mu\alpha_2.\ [\alpha \mapsto \mu\alpha_1.\ D]\ B'$.
  - If we apply rule S-REC to the goal, we get: $\forall n,\ \Gamma_1, \Gamma_2, \alpha_2 \vdash [\alpha_2 \mapsto ([\alpha \mapsto \mu\alpha_1.\ C]\ A')]^n\ ([\alpha \mapsto \mu\alpha_1.\ C]\ A') \leq [\alpha_2 \mapsto ([\alpha \mapsto \mu\alpha_1.\ D]\ B')]^n\ ([\alpha \mapsto \mu\alpha_1.\ D]\ B')$.
  - We rewrite the goal above, getting: $\forall n,\ \Gamma_1, \Gamma_2, \alpha_2 \vdash [\alpha \mapsto \mu\alpha_1.\ C][\alpha_2 \mapsto A']^n\ A' \leq [\alpha \mapsto \mu\alpha_1.\ D][\alpha_2 \mapsto B']^n\ B'$.
  - The induction hypothesis is complex, so we write it here explicitly for readability of the proof: $\forall n,\ (\forall n',\ \Gamma_1, \alpha_1, \Gamma_2 \vdash [\alpha_1 \mapsto C]^n[\alpha \mapsto \alpha_1][\alpha_2 \mapsto A']^{n'}\ A' \leq [\alpha_1 \mapsto D]^n[\alpha \mapsto \alpha_1][\alpha_2 \mapsto B']^{n'}\ B') \Rightarrow \Gamma_1, \alpha_2, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha_1.\ C][\alpha_2 \mapsto A']^n\ A' \leq [\alpha \mapsto \mu\alpha_1.\ D][\alpha_2 \mapsto B']^n\ B'$.
  - By applying context permutation and induction hypothesis to the goal, we get: $\forall n,\ \forall n',\ \Gamma_1, \alpha_1, \Gamma_2 \vdash [\alpha_1 \mapsto C]^n[\alpha \mapsto \alpha_1][\alpha_2 \mapsto A']^{n'}\ A' \leq [\alpha_1 \mapsto D]^n[\alpha \mapsto$

$\alpha_1][\alpha_2 \mapsto B']^{n'}\ B'$, which can be proven by the inversion of fourth premise due to the fact that substitution is commutative.

□

Lemma 7 captures the idea of finite approximation. It relates the boundless unfolding with limited unfolding. This lemma is a generalization of the unfolding lemma, and when $A = C$ and $B = D$, one easily obtains the unfolding lemma.

Lemma 8. Unfolding Lemma.

$$\text{If } \Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B \text{ then } \Gamma \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B.$$

3.4 Type Safety

In this section, we show that our system is type sound by proving preservation and progress.

3.4.1 Typing and reduction rules

As the top of Figure 3.4 shows, the typing rules are quite standard. Noteworthy are the rules involving recursive types. Rule TYPING-UNFOLD reveals that if e has type $\mu\alpha.\ A$ then, after unfolding, its type becomes $[\alpha \mapsto \mu\alpha.\ A]\ A$. Rule TYPING-FOLD says if e has type $[\alpha \mapsto \mu\alpha.\ A]\ A$, after folding, its type becomes $\mu\alpha.\ A$, with an additional type well-formedness check on $\mu\alpha.\ A$. The two constructs establish an isomorphism, which is used to deal with expressions with iso-recursive types. The last rule is the standard subsumption rule (rule TYPING-SUB).

The bottom of Figure 3.4 shows the reduction rules, which are also quite standard. We only focus on the last three rules involving recursive types. Rule STEP-FLD cancels a pair of unfold and fold. Note that the two types A and B are not necessarily the same. The last two rules (rule STEP-UNFOLD and rule STEP-FOLD) simply reduce the inner expressions for unfold's and fold's.

3.4.2 Type soundness

In this subsection, we briefly illustrate how to prove type-soundness. The technique is mostly conventional, except for the fundamental use of the unfolding lemma in the preservation proof (via Lemma 10). Firstly, we need a conventional substitution lemma to deal with beta reduction in preservation:

Lemma 9. Substitution lemma. If $\Gamma_1, x : B, \Gamma_2 \vdash e : A$ and $\Gamma_2 \vdash e' : B$ then $\Gamma_1, \Gamma_2 \vdash [x \mapsto e']\ e : A$.

Then we show how the unfolding lemma is used in the proof on type soundness, via an inversion of typing lemma for fold expressions:

Lemma 10. Inversion of typing for fold expressions: If $\Gamma \vdash \text{fold}\ [A]\ e : S$ and $\Gamma \vdash S \leq \mu\alpha.\ B$, then $\exists T,\ \Gamma \vdash e : [\alpha \mapsto \mu\alpha.\ T]\ T \wedge \Gamma \vdash [\alpha \mapsto \mu\alpha.\ T]\ T \leq [\alpha \mapsto \mu\alpha.\ B]\ B$.

$\boxed{\Gamma \vdash e : A}$ *(Typing)*

$$\textsc{typing-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{i} : \mathsf{nat}} \qquad \textsc{typing-var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{typing-abs}\ \frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash \lambda x : A_1.\ e : A_1 \to A_2}$$

$$\textsc{typing-unfold}\ \frac{\Gamma \vdash e : \mu\alpha.\ A}{\Gamma \vdash \mathsf{unfold}\ [\mu\alpha.\ A]\ e : [\alpha \mapsto \mu\alpha.\ A]\ A} \qquad \textsc{typing-fold}\ \frac{\Gamma \vdash e : [\alpha \mapsto \mu\alpha.\ A]\ A \qquad \Gamma \vdash \mu\alpha.\ A}{\Gamma \vdash \mathsf{fold}\ [\mu\alpha.\ A]\ e : \mu\alpha.\ A}$$

$$\textsc{typing-app}\ \frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2} \qquad \textsc{typing-sub}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}$$

$\boxed{e_1 \hookrightarrow e_2}$ *(Reduction)*

$$\textsc{step-beta}\ \frac{}{(\lambda x : A.\ e_1)\ v_2 \hookrightarrow [x \mapsto v_2]\ e_1} \qquad \textsc{step-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2} \qquad \textsc{step-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\ e_2 \hookrightarrow v_1\ e_2'}$$

$$\textsc{step-fld}\ \frac{}{\mathsf{unfold}\ [A]\ (\mathsf{fold}\ [B]\ v) \hookrightarrow v} \qquad \textsc{step-unfold}\ \frac{e \hookrightarrow e'}{\mathsf{unfold}\ [A]\ e \hookrightarrow \mathsf{unfold}\ [A]\ e'} \qquad \textsc{step-fold}\ \frac{e \hookrightarrow e'}{\mathsf{fold}\ [A]\ e \hookrightarrow \mathsf{fold}\ [A]\ e'}$$

Figure 3.4: Typing and reduction rules.

Proof. Do induction on $\Gamma \vdash \mathsf{fold}\ [A]\ e : S$.

- Rule TYPING-FOLD: the premises become $\Gamma \vdash e : [\alpha \mapsto \mu\alpha.\ A']\ A'$ (assume $A = \mu\alpha.\ A'$) and $\Gamma \vdash \mu\alpha.\ A' \leq \mu\alpha.\ B$. In such situation, let $T = A'$, we achieve the goal by applying the unfolding lemma (Lemma 8).
- Rule TYPING-SUB: trivial by applying transitivity (Theorem 6).

□

Finally, we can proceed to the preservation and progress theorems, and the proof strategy is quite standard.

Theorem 11. Preservation.

$$\text{If } \Gamma \vdash e : A \text{ and } e \hookrightarrow e' \text{ then } \Gamma \vdash e' : A.$$

Proof. By induction on $\Gamma \vdash e : A$. Most cases are trivial or standard, except for

- Rule TYPING-UNFOLD. In this case, e is decomposed into $\mathsf{unfold}\ [\mu\alpha.\ A]\ e$, and our goal is to prove $\Gamma \vdash e' : [\alpha \mapsto \mu\alpha.\ A]\ A$.

  By inversion on $\mathsf{unfold}\ [\mu\alpha.\ A]\ e \hookrightarrow e'$, we will get two sub-cases.

  - The case for rule STEP-UNFOLD is trivial: e' continues to decompose into $\mathsf{unfold}\ [\mu\alpha.\ A]\ e'$. By applying rule TYPING-UNFOLD and induction hypothesis, we achieve the goal.

– As for case involving rule STEP-FLD, the first premise becomes $\Gamma \vdash \text{fold } [A'] \; v : \mu\alpha.\, A$. Then we do the inversion on the first premise again, get two sub-cases. The first case is same as the goal. The second case, raised by rule TYPING-SUB, needs some extra work: what we get now are $\Gamma \vdash \text{fold } [A'] \; v : S$ and $\Gamma \vdash S \leq \mu\alpha.\, A$. Then we apply Lemma 10 (where the unfolding lemma is used) and rule TYPING-SUB to achieve the goal.

□

Theorem 12. Progress.

$$\text{If } \vdash e : A \text{ then } e \text{ is a value or exists } e', e \hookrightarrow e'.$$

Proof. By induction on $\vdash e : A$. □

3.5 Mechanized Proofs

The folder *coq_theory* includes all the Coq proofs about STLC extended with iso-recursive subtyping, which is the calculus described in this chapter.

3.5.1 Definitions

All the definitions in the Chapter 3 can be found in the file *Rules.v*. Table 3.1 shows the correspondence of definitions between the paper and the Coq artifacts. The file *Rules.v* contains the definitions for our type system. It has definitions of well-formedness, subtyping, typing, and reduction.

For encoding variables and binders, we use the locally nameless representation to express all the types and terms. In the paper, we use only *substitution* to represent *unfolding* of a recursive type. In the Coq proof, due to the use of the locally nameless representation, we also use of *opening* operation on pre-terms [11]. Furthermore, in the paper, we always use the same notation for well-formedness with rule WFT-REC, rule WFT-INF. In the Coq formalization, we have two distinct definitions of well-formedness, which are proved to be equivalent.

3.5.2 Lemmas and theorems

Table 3.2 shows the descriptions for all the proof scripts in Chapter 3. For succinctness, we briefly describe the important lemmas and theorems.

Table 3.1: Paper-to-proofs correspondence guide in Chapter 3.

Definition	File	Name in Coq	Notation
Well-formed Type (Figure 3.3)	Rules.v	WFA E A	$\Gamma \vdash A$
Well-formed Type (Definition 2)	Rules.v	WFS E A	$\Gamma \vdash A$
Declarative Subtyping (Figure 3.3)	Rules.v	Sub E A B	$\Gamma \vdash A \leq B$
Typing (Figure 3.4)	Rules.v	typing E e A	$\Gamma \vdash e : A$
Reduction (Figure 3.4)	Rules.v	step e1 e2	$e_1 \hookrightarrow e_2$

Table 3.2: Descriptions for the proof scripts in Chapter 3.

Theorems	Description	Files	Name in Coq
Theorem 5	Reflexivity	FiniteUnfolding.v	refl
Theorem 6	Transitivity	FiniteUnfolding.v	Transitivity
Lemma 8	Unfolding Lemma	FiniteUnfolding.v	unfolding_lemma
Theorem 11	Preservation	Typesafety.v	preservation
Theorem 12	Progress	Typesafety.v	progress

Chapter 4

Algorithmic Subtyping for Iso-Recursive Subtyping

In Chapter 3 we introduced a declarative formulation of subtyping with recursive types. Unfortunately, such formulation is not directly implementable since the rule of subtyping for recursive types checks against an infinite number of conditions (that all finite unfoldings are subtypes). In this chapter, we will present two sound and complete algorithmic formulations of subtyping. These formulations replace the declarative rule S-REC by rules based on double unfoldings. A first rule, which we call the double unfolding rule, unfolds the recursive types 1-time and 2-times, respectively. The double unfolding rule relates to different subtyping formulations in this thesis, playing a significant role as a hub, as shown in Figure 5.4 in the next Chapter. We then give another algorithmic variant, using the nominal unfolding rule, and finally prove our subtyping rules for iso-recursive types are decidable.

4.1 Overview

An infinite amount of conditions is impossible to check algorithmically. Therefore, we must find alternative formulations for rule S-REC that are algorithmic for implementations. As we will show in Chapter 5, a suitable formulation with iso-recursive Amber rules is equivalent to our declarative specification. Thus, the Amber rules can, in principle, serve as a foundation for an implementation. However, there are reasons to seek for alternative algorithmic rules. Most importantly, as we have argued in Chapter 2.1.4, the Amber rules are hard to work with in proofs and metatheory. Therefore, to provide a detailed account of the metatheory for iso-recursive subtyping we propose alternative algorithmic definitions for subtyping of recursive types. The new formulations of subtyping have important advantages over the Amber rules: the new rules are more modular; they do not require reflexivity to be built-in; and transitivity and various other lemmas are easier to prove. Furthermore, in Chapter 5, we prove that the new rules are also sound and complete with respect to the declarative specification of iso-recursive subtyping and the Amber rules.

4.1.1 Double unfoldings

It turns out that we only need to check 1-time and 2-times finite unfoldings to obtain an algorithmic formulation that is *sound*, *complete* and *decidable* with respect to the declarative formulation of subtyping. We can informally explain why 1-time and 2-times finite unfoldings are enough by looking again at the counter-example in Chapter 2.1.1. The 2-times finite unfolding for the example is:

$$\alpha \vdash (\alpha \to \mathsf{nat}) \to \mathsf{nat} \leq (\alpha \to \top) \to \top$$

When a recursive type variable in a negative position is unfolded twice, the types in the corresponding positive positions (i.e. the nat and $\top$) will now appear in both negative and positive positions. In turn, the subtyping relation now has to check both that $\mathsf{nat} \leq \top$ (which is valid), and $\top \leq \mathsf{nat}$ (which is invalid). Thus, the 2-times finite unfolding fails. In general, more finite unfoldings (3-times, 4-times, etc.) will only repeat the same checks that are done by the 1-time and 2-times finite unfolding, thus not contributing anything new to the subtyping check. Thus, the rule that we employ in the algorithmic formulation is the so-called *double unfolding* rule:

$$\frac{\Gamma, \alpha \vdash A \leq B \qquad \Gamma, \alpha \vdash [\alpha \mapsto A]\, A \leq [\alpha \mapsto B]\, B}{\Gamma \vdash \mu\alpha.\, A \leq \mu\alpha.\, B}\ \textsc{S-double}$$

With this rule one may wonder if we can just check the 2-times finite unfolding (and do not do the 1-time finite unfolding check). Unfortunately this would lead to an unsound rule, as the following counter-example illustrates:

$$\mu\alpha.\, \mathsf{nat} \to \alpha \not\leq \mu\alpha.\, \mathsf{nat} \to \mathsf{nat} \to \top$$

This statement should fail because it violates the *unfolding lemma*:

$$\mathsf{nat} \to (\mu\alpha.\, \mathsf{nat} \to \alpha) \not\leq \mathsf{nat} \to \mathsf{nat} \to \top$$

But the 2-times finite unfolding for this example ($\mathsf{nat} \to \mathsf{nat} \to \alpha \leq \mathsf{nat} \to \mathsf{nat} \to \top$) is a valid subtyping statement! By checking only the 2-times finite unfolding, the subtyping statement is wrongly accepted. We must also check the 1-time finite unfolding ($\mathsf{nat} \to \alpha \not\leq \mathsf{nat} \to \mathsf{nat} \to \top$), which fails and is the reason why the double unfolding rule rejects this example.

4.1.2 The spurious subtyping problem

The double unfolding rule is interesting because it directly relates to the declarative formulation using finite unfoldings. However, the double unfolding rules have exponential time complexity due to the two premises for both (1-time and 2-times) finite unfoldings. At first, the 1-time finite unfolding appears unnecessary, since the 2-times unfolding seems to do all the checks of the 1-time finite unfolding. Unfortunately, as our previous counter-example has

shown, the 1-time finite unfolding check cannot be avoided, due to some spurious subtyping that exists when using only the 2-times finite unfolding.

Type unsoundness with 2-times unfolding only Some reader might ask if we can accept the subtyping rule with 2-times finite unfoldings only, like

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A]\ A \leq [\alpha \mapsto B]\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}$$

As we have seen in Chapter 4.1.1, such a rule does not necessarily satisfy the unfolding lemma. Unfortunately there is a more serious problem with the rule: it is not type sound.

For convenience, let's denote $A := \mu\alpha.\ \mathsf{nat} \to \alpha$ and $B := \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \top$. Assume that $A \leq B$ holds, but $\mathsf{nat} \to A$ is not subtype of $\mathsf{nat} \to \mathsf{nat} \to \top$ (the unfolding lemma fails).

Then assume that we have the expression:

$$e = \mathsf{fix}(x : A).\ (\mathsf{fold}\ [A]\ (\lambda y : \mathsf{nat}.\ x))$$

Although the calculi we present do not include fixpoints, those can be easily added (the calculus in Chapter 7 has fixpoints), with the rules:

Definition 13. The typing rule for fixpoint is:

$$\frac{\Gamma, x : A \vdash e : A}{\Gamma \vdash \mathsf{fix}\ x : A.\ e : A}\ \textsc{typing-fix}$$

Definition 14. The reduction rule for fixpoint is:

$$\frac{}{\mathsf{fix}\ x : A.\ e \hookrightarrow [x \mapsto (\mathsf{fix}\ x : A.\ e)]\ e}\ \textsc{step-fixpoint}$$

Now, consider such an expression $(\mathsf{unfold}\ [B]\ e)$. Firstly, we show that it can be type-checked:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x : A \vdash \lambda y : \mathsf{nat}.\ x : \mathsf{nat} \to A}{x : A \vdash \mathsf{fold}\ [A]\ (\lambda y : \mathsf{nat}.\ x) : A}\ \textsc{typing-fold} \qquad A \leq B}{x : A \vdash \mathsf{fold}\ [A]\ (\lambda y : \mathsf{nat}.\ x) : B}\ \textsc{typing-sub}}{\vdash \mathsf{fix}(x : A).\ (\mathsf{fold}\ [A]\ (\lambda y : \mathsf{nat}.\ x)) : B}\ \textsc{typing-fix}}{\vdash e : B}}{\vdash \mathsf{unfold}\ [B]\ e : \mathsf{nat} \to \mathsf{nat} \to \top}\ \textsc{typing-unfold}$$

Next, we show that $(\mathsf{unfold}\ [B]\ e)$ can reduce to $(\mathsf{unfold}\ [B]\ (\mathsf{fold}\ [A]\ (\lambda y : \mathsf{nat}.\ e)))$:

$$\dfrac{\dfrac{}{\mathsf{fix}(x : A).\ (\mathsf{fold}\ [A]\ (\lambda y : \mathsf{nat}.\ x)) \rightsquigarrow \mathsf{fold}\ [A]\ (\lambda y : \mathsf{nat}.\ e)}\ \textsc{step-fixpoint}}{\mathsf{unfold}\ [B]\ e \rightsquigarrow \mathsf{unfold}\ [B]\ (\mathsf{fold}\ [A]\ (\lambda y : \mathsf{nat}.\ e))}\ \textsc{step-unfold}$$

and then can reduce to $(\lambda y : \mathsf{nat}.\ e)$:

$$\frac{}{\mathsf{unfold}\ [B]\ (\mathsf{fold}\ [A]\ (\lambda y : \mathsf{nat}.\ e)) \rightsquigarrow \lambda y : \mathsf{nat}.\ e}\ \textsc{step-fld}$$

If the preservation theorem holds, we wish that $(\lambda y : \text{nat}.\ e)$ also has same type as $(\text{unfold}\ [B]\ e)$. However, the type of $(\lambda y : \text{nat}.\ e)$ isn't $\text{nat} \to \text{nat} \to \top$. Thus, the preservation theorem is false.

4.1.3 Nominal unfolding

The double unfolding rule is costly. In an implementation, there are potentially some approaches to avoid the cost of the extra 1-time finite unfolding check. For example, we can store the result of 1-time finite unfolding during subtype checking, and reuse that result as part of subtype checking of the double unfoldings. This would avoid recomputation and lead to a more efficient algorithm. However, it would be nicer to address this issue of the double unfoldings in the formalism itself.

For avoiding the 1-time finite unfolding in the double unfolding rule, we propose a variant of the rule. Having understood the nature of the spurious subtyping problem that appeared in our counter-example using only 2-times finite unfoldings, the key idea to solve the problem is simple. We need to track the substituted types during double unfoldings to avoid accidental subtyping. Our approach is to add an extra fresh label to exposure the substitution in the 2-times finite unfolding. This regulates the structure of the derivation tree. Formally, our nominal unfolding rule is:

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}\ \text{S-FNOMINAL}$$

The nominal unfolding rules use *labelled types* A^{α}, to ensure that we only compare types that arise from unfolding substitutions with related unfolded types. Labelled types are a syntactic devise used to prevent accepting subtyping statements such as $\mu\alpha.\ \text{nat} \to \alpha \leq \mu\alpha.\ \text{nat} \to \text{nat} \to \top$, which would be unsound in an iso-recursive formulation. They also provide a distinct nominal identity to the recursive types being compared, so that they cannot be compared with other unrelated recursive types. Note that, in the rule S-NOMINAL, we have reused the recursive variable name for the label α. However, we used the red color to distinguish the label and type variable names. Labels α should be the same in both substitutions, and distinct from any other labels and bound variables used elsewhere. Since in the paper presentation we use a nominal approach to represent binders, the label α should be interpreted as some unique freshly generated name[1].

Compared to the double unfolding rule, our nominal unfolding rule only has one subtyping check. More importantly, it avoids the spurious subtyping problem. In our new nominal unfolding rule, we do not need the extra check for the 1-time finite unfolding (checking $\Gamma, \alpha \vdash A \leq B$). The derivation tree below reflects the change for our simpler counter-example of the double unfolding rule (without the extra 1-time finite unfolding check):

[1] In our Coq formalization we use a locally nameless representation [11], which distinguishes free and bound variables naturally. With a locally nameless representation we can reuse the free variable name α for the label α.

$$\frac{\dfrac{\mathsf{nat} \leq \mathsf{nat} \qquad (\mathsf{nat} \to \alpha)^{\alpha} \leq \mathsf{nat} \to \top \text{ (fails!)}}{\mathsf{nat} \to (\mathsf{nat} \to \alpha)^{\alpha} \leq \mathsf{nat} \to \mathsf{nat} \to \top}}{\mu\alpha.\ \mathsf{nat} \to \alpha \leq \mu\alpha.\ \mathsf{nat} \to \mathsf{nat} \to \top}$$

The presence of the extra label means that we now get $(\mathsf{nat} \to \alpha)^{\alpha} \leq \mathsf{nat} \to \top$ (which fails) instead of $\mathsf{nat} \to \alpha \leq \mathsf{nat} \to \top$ (which succeeds). In other words, the presence of the nominal label avoids the need for the extra 1-time finite unfolding check to rule out the counter-example.

4.2 Double Unfoldings

In this section, we present a variant of the calculus introduced in Chapter 3. The key difference is that instead of using the declarative rule for recursive subtyping, we employ a rule with double unfoldings. Our formalization includes syntax, well-formedness, subtyping and corresponding theorems.

4.2.1 Syntax, well-formedness and subtyping

The syntax and well-formedness of the double unfoldings share the same definitions as the declarative system presented in Chapter 3.2.

Well-Formedness In the algorithmic version, we use $\Gamma \vdash A$ to represent that A is well-formed. The rules of $\Gamma \vdash A$ are the same as the top of Figure 3.3. Similarly to Chapter 3.2, we define an alternative variant of well-formedness with the rule WFT-RECUR to give us better induction principles for the proofs.

Definition 15. Rule WFT-RECUR is defined as:

$$\frac{\Gamma, \alpha \vdash A \qquad \Gamma, \alpha \vdash [\alpha \mapsto A]\, A}{\Gamma \vdash \mu\alpha.\ A}\ \text{WFT-RECUR}$$

Subtyping Figure 4.1 shows the algorithmic subtyping judgment. All the rules, except the one for recursive types, remain the same as the declarative system. In algorithmic subtyping, rule SA-REC states that two recursive types are subtypes when: 1) their bodies are subtypes; and 2) unfolding the bodies one additional time preserves subtyping. In other words, checking 1-time and 2-times finite unfoldings rather than all finite unfoldings is sufficient.

4.2.2 Reflexivity, transitivity and completeness

Our algorithmic subtyping simply relaxes the condition for recursive types while keeping the judgment form. Therefore, regularity, reflexivity and transitivity are easy to prove using similar techniques to those used in the declarative system.

Lemma 16. Regularity: If $\Gamma \vdash_a A \leq B$ then $\vdash \Gamma$ and $\Gamma \vdash A$ and $\Gamma \vdash B$.

$\boxed{\Gamma \vdash_a A \leq B}$ *(Algorithmic Subtyping)*

$$\text{SA-NAT}\ \frac{\vdash \Gamma}{\Gamma \vdash_a \mathsf{nat} \leq \mathsf{nat}} \qquad \text{SA-TOP}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash_a A \leq \top} \qquad \text{SA-VAR}\ \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash_a \alpha \leq \alpha}$$

$$\text{SA-ARROW}\ \frac{\Gamma \vdash_a B_1 \leq A_1 \qquad \Gamma \vdash_a A_2 \leq B_2}{\Gamma \vdash_a A_1 \to A_2 \leq B_1 \to B_2} \qquad \text{SA-REC}\ \frac{\Gamma, \alpha \vdash_a A \leq B \qquad \Gamma, \alpha \vdash_a [\alpha \mapsto A]\, A \leq [\alpha \mapsto B]\, B}{\Gamma \vdash_a \mu\alpha.\, A \leq \mu\alpha.\, B}$$

Figure 4.1: Algorithmic subtyping.

Theorem 17. Reflexivity.

$$\text{If } \vdash \Gamma \text{ and } \Gamma \vdash A \text{ then } \Gamma \vdash_a A \leq A.$$

Theorem 18. Transitivity.

$$\text{If } \Gamma \vdash_a A \leq B \text{ and } \Gamma \vdash_a B \leq C \text{ then } \Gamma \vdash_a A \leq C.$$

Proof. Because $\Gamma \vdash_a A \leq B$, we know that type B is well-formed by Lemma 16. Then proceed by induction on $\Gamma \vdash B$. □

Note that, like the declarative system (and unlike the Amber rules), the transitivity proof is very simple with the double unfolding rule. The completeness of algorithmic subtyping is obvious, since the declarative system has the same conditions of the algorithmic system (plus a few more).

Theorem 19. Completeness of algorithmic subtyping.

$$\text{If } \Gamma \vdash A \leq B \text{ then } \Gamma \vdash_a A \leq B.$$

Proof. By induction on $\Gamma \vdash A \leq B$. Then for recursive case, we know that for all n, the subtyping relation is holds after unfolding n times. Choosing $n = 1$ and $n = 2$, we know that the two premises of algorithmic recursive subtyping are satisfied. □

4.3 The Soundness Theorem

The real challenge is the soundness of the algorithmic specification with respect to the declarative system. For soundness, we wish to prove that:

$$\text{If } \Gamma \vdash_a A \leq B \text{ then } \Gamma \vdash A \leq B.$$

The key problem is to show that finitely unfolding only one and two times is sufficient to guarantee that all finite unfoldings are sound. Although it is easy to give an informal argument

as to why this is the case, as we did in Chapter 4.1, formalizing this argument is a whole different matter.

4.3.1 Finding the right generalization for soundness

The key idea to prove that 1-time and 2-times finite unfolding implies n-times finite unfolding is to capture this informal idea formally as a lemma:

$$\Gamma \vdash A \leq B \quad \wedge \quad \Gamma \vdash [\alpha \mapsto A]\, A \leq [\alpha \mapsto B]\, B \quad \Rightarrow \quad \Gamma \vdash [\alpha \mapsto A]^n\, A \leq [\alpha \mapsto B]^n\, B.$$

As we shall see this lemma is true but, unfortunately, it cannot be proved directly. The obvious attempt would be to do induction on $\Gamma \vdash A \leq B$. The essential problem with such an approach is that we wish to analyze the different subcases for A and B, but we still want to use the original A and B in the substitutions. For instance, suppose that we have $A := \mathsf{nat} \to A_1 \to A_2$ and $B := \mathsf{nat} \to B_1 \to B_2$. Here $A_1 \to A_2$ and $B_1 \to B_2$ are contained in the type A and B. Now consider the case for function types $\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2$, which would occur as a subcase in the proof. What we would like to have is the conclusion

$$\Gamma \vdash [\alpha \mapsto A]^n\, (A_1 \to A_2) \leq [\alpha \mapsto B]^n\, (B_1 \to B_2)$$

However, what we get instead is

$$\Gamma \vdash [\alpha \mapsto (A_1 \to A_2)]^n\, (A_1 \to A_2) \leq [\alpha \mapsto (B_1 \to B_2)]^n\, (B_1 \to B_2)$$

In other words, what gets substituted are not the original types A and B, but only a part of those types ($A_1 \to A_2$ and $B_1 \to B_2$) that is being considered by the current case. Therefore, it is clear that we need some generalization of this lemma. A first idea is to generalize it as follows:

$$\begin{array}{rc} & \Gamma \vdash A \leq B \quad \wedge \quad \Gamma \vdash C \leq D \quad \wedge \quad \Gamma \vdash [\alpha \mapsto C]\, A \leq [\alpha \mapsto D]\, B \\ \Rightarrow & \Gamma \vdash [\alpha \mapsto C]^n\, A \leq [\alpha \mapsto D]^n\, B. \end{array}$$

Now it is possible to do induction on $\Gamma \vdash A \leq B$ without affecting the substituted types. However, this lemma is *false*. A counter-example is:

$$\begin{array}{rc} & \Gamma \vdash \top \to \alpha \leq \mathsf{nat} \to \alpha \quad \wedge \quad \Gamma \vdash \alpha \to \mathsf{nat} \leq \alpha \to \top \\ & \wedge \quad \Gamma \vdash \top \to \alpha \to \mathsf{nat} \leq \mathsf{nat} \to \alpha \to \top \\ \not\Rightarrow & \Gamma \vdash \top \to (\alpha \to \mathsf{nat}) \to \mathsf{nat} \leq \mathsf{nat} \to (\alpha \to \top) \to \top. \end{array}$$

In this counter-example we choose $n = 2$. All the premises are satisfied, but the conclusion is false. Note that in the conclusion, because of the contravariance of function subtyping, we eventually require that $\Gamma \vdash \alpha \to \top \leq \alpha \to \mathsf{nat}$, which is clearly false.

By further analyzing the counter-example, we can see that the influence of contravariance on variables is not reflected in such a lemma. Therefore, our generalized soundness lemma

should deal with type variables at contravariant positions and covariant positions respectively, but under the same pattern. In other words we need a pair of lemmas: one to deal with covariance, and another to deal with contravariance.

4.3.2 The generalized lemma

Learning from the lessons of the failed attempts at soundness we reach to the following lemma, which holds:

Lemma 20. If,

1. $\Gamma \vdash A \leq B$;
2. $\Gamma \vdash C \leq D$;
3. $\Gamma \vdash [\alpha \mapsto C]^n\ C \leq [\alpha \mapsto D]^n\ D$.

then

1. $\Gamma \vdash [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$ implies $\Gamma \vdash [\alpha \mapsto C]^{n+1}\ A \leq [\alpha \mapsto D]^{n+1}\ B$ and
2. $\Gamma \vdash [\alpha \mapsto D]\ A \leq [\alpha \mapsto C]\ B$ implies $\Gamma \vdash [\alpha \mapsto D]^{n+1}\ A \leq [\alpha \mapsto C]^{n+1}\ B$.

Proof. By induction on $\Gamma \vdash A \leq B$.

- Case rule S-VAR: In such case $A = B = \beta$. If $\beta \neq \alpha$, we prove the goal trivially. Otherwise,
  - ⋆ Goal (1): We want to prove $\Gamma \vdash [\alpha \mapsto C]^n\ C \leq [\alpha \mapsto D]^n\ D$, which can be obtained from premise (3).
  - ⋆ Goal (2), We have premises $\Gamma \vdash C \leq D$ by premise (2) and $\Gamma \vdash D \leq C$ from the condition of goal (2), thus $C = D$ by Lemma 21. Goal (2) is proven by reflexivity.
- Case rule S-ARROW: In such case $A = A_1 \to A_2$ and $B = B_1 \to B_2$.
  - ⋆ Goal (1):

    We need to prove $\Gamma \vdash [\alpha \mapsto C]^{n+1}\ (A_1 \to A_2) \leq [\alpha \mapsto D]^{n+1}\ (B_1 \to B_2)$, which can be rewritten as $\Gamma \vdash ([\alpha \mapsto C]^{n+1}\ A_1) \to ([\alpha \mapsto C]^{n+1}\ A_2) \leq ([\alpha \mapsto D]^{n+1}\ B_1) \to ([\alpha \mapsto D]^{n+1}\ B_2)$. By construction, we need to prove $\Gamma \vdash [\alpha \mapsto D]^{n+1}\ B_1 \leq [\alpha \mapsto C]^{n+1}\ A_1$ and $\Gamma \vdash [\alpha \mapsto C]^{n+1}\ A_2 \leq [\alpha \mapsto D]^{n+1}\ B_2$. The former one can be proved by using the induction hypothesis arising from goal (2), while the latter one can be proved by using the induction hypothesis arising from goal (1).
  - ⋆ Goal (2):

    We need to prove $\Gamma \vdash [\alpha \mapsto D]^{n+1}\ (A_1 \to A_2) \leq [\alpha \mapsto C]^{n+1}\ (B_1 \to B_2)$, which can be rewritten as $\Gamma \vdash ([\alpha \mapsto D]^{n+1}\ A_1) \to ([\alpha \mapsto D]^{n+1}\ A_2) \leq ([\alpha \mapsto C]^{n+1}\ B_1) \to ([\alpha \mapsto C]^{n+1}\ B_2)$. By construction, we need to prove $\Gamma \vdash [\alpha \mapsto C]^{n+1}\ B_1 \leq [\alpha \mapsto D]^{n+1}\ A_1$ and $\Gamma \vdash [\alpha \mapsto D]^{n+1}\ A_2 \leq [\alpha \mapsto C]^{n+1}\ B_2$. The

former one can be proved by using the induction hypothesis arising from goal (1), while the latter one can be proved by using the induction hypothesis arising from goal (2).

- Case rule S-REC: Now we assume $A = \mu\alpha'.\ A'$ and $B = \mu\alpha'.\ B'$. Since in such case, we do not need to consider the contravariance, we will just show how to prove goal (1). Goal (2) can be proved using the same approach.
  - The condition arising from the goal (1) becomes $\Gamma \vdash [\alpha \mapsto C]\ \mu\alpha'.\ A \leq [\alpha \mapsto D]\ \mu\alpha'.\ B$. After inversion, we get $\forall n', \Gamma \vdash [\alpha' \mapsto A']^{n'}[\alpha \mapsto C]\ A' \leq [\alpha' \mapsto B']^{n'}[\alpha \mapsto D]\ B'$, which can be rewritten as $\forall n', \Gamma \vdash [\alpha \mapsto C][\alpha' \mapsto A']^{n'}\ A' \leq [\alpha \mapsto D][\alpha' \mapsto B']^{n'}\ B'$.
  - The goal now is $\Gamma \vdash [\alpha \mapsto C]^{n+1}\ \mu\alpha'.\ A' \leq [\alpha \mapsto D]^{n+1}\ \mu\alpha'.\ B'$, which can be rewritten as $\Gamma \vdash \mu\alpha'.\ [\alpha \mapsto C]^{n+1}\ A' \leq \mu\alpha'.\ [\alpha \mapsto D]^{n+1}\ B'$.
  - Applying rule S-REC on the goal, we get $\forall n',\ \Gamma, \alpha' \vdash [\alpha' \mapsto A']^{n'}[\alpha \mapsto C]^{n+1}\ A' \leq [\alpha' \mapsto B']^{n'}[\alpha \mapsto D]^{n+1}\ B'$, which can be rewritten as $\forall n',\ \Gamma, \alpha' \vdash [\alpha \mapsto C]^{n+1}[\alpha' \mapsto A']^{n'}\ A' \leq [\alpha \mapsto D]^{n+1}[\alpha' \mapsto B']^{n'}\ B'$.
  - Finally, we apply the induction hypothesis, to prove goal (1).

□

Compared with our last failed attempt, there is an extra condition (condition 3). More importantly, there are now two conclusions. These conclusions basically express two different lemmas. One lemma, with all the conditions and conclusion (1), and another lemma with all conditions and conclusion (2). Conclusion (1) covers covariant uses of the lemma, whereas conclusion (2) covers contravariant uses of the lemma. Note that when we apply the lemma in our soundness theorem, we have that $A = C$ and $B = D$. Those types will then become different as the subcases of type A and B are processed. For covariant cases, A is a portion of the type C, and B is a portion of the type D. Conclusion (1) covers this, and we can see that we are substituting C in A and D in B. However, the contravariance of function types will flip the input types being checked for subtyping. This means that in effect, A is now a portion of D (in a contravariant position in D) and B is a portion of C (in a contravariant position in C). Goal (2) captures such nuance and provides a formulation for the lemma that deals with subparts of C and D, which are in contravariant positions.

The proof of Lemma 20 relies on the following property of the subtyping relation:

Lemma 21. Antisymmetry of declarative subtyping: If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq A$ then $A = B$.

Also, from Lemma 20, we now can prove:

Lemma 22. If $\Gamma \vdash A \leq B$ and $\Gamma \vdash [\alpha \mapsto A]\ A \leq [\alpha \mapsto B]\ B$, then $\forall n,\ \Gamma \vdash [\alpha \mapsto A]^n\ A \leq [\alpha \mapsto B]^n\ B$.

Proof. Do induction on n. For the base case, we simply apply the premise (1). For the induction case, we apply Lemma 20 with $C = A$ and $D = B$, then apply induction hypothesis. □

The form of Lemma 22 is close to the shape of the infinite unfolding rule (rule S-REC) for recursive types. Finally, we can prove the soundness theorem:

Theorem 23. Soundness of algorithmic subtyping.

$$\text{If } \Gamma \vdash_a A \leq B \text{ then } \Gamma \vdash A \leq B.$$

4.4 The Unfolding Lemma for the Double Unfolding Rules

In Chapter 3.3.3, we showed how to prove the unfolding lemma for the declarative system. It turns out that the unfolding lemma can also be proved relatively easily for the algorithmic system using a technique similar to that employed in the proof of soundness in Chapter 4.3. A direct proof of the unfolding lemma is useful for language designers wishing to skip the declarative system, and formulate only an algorithmic version.

Lemma 20 provides an interesting (and necessary) lemma for proving soundness between double and finite unfoldings. For that lemma a key insight is that we need two forms: one for dealing with contravariant cases, and another to deal with covariant cases. Inspired by this insight, we are able to prove the unfolding lemma directly for the double unfolding rules, using a similar technique. Firstly we need a lemma similar to Lemma 21, but for the algorithmic relation:

Lemma 24. Antisymmetry of algorithmic subtyping: If $\Gamma \vdash_a A \leq B$ and $\Gamma \vdash_a B \leq A$ then $A = B$.

Then we can formulate the generalized lemma that is needed to prove the unfolding lemma as follows:

Lemma 25. If

1. $\Gamma_1, \alpha, \Gamma_2, \vdash_a A \leq B$;
2. $\Gamma_1, \alpha, \Gamma_2, \vdash_a C \leq D$;
3. $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\, C \leq \mu\alpha.\, D$;

then

1. $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto C]\, A \leq [\alpha \mapsto D]\, B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, C]\, A \leq [\alpha \mapsto \mu\alpha.\, D]\, B$ and
2. $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto D]\, A \leq [\alpha \mapsto C]\, B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, D]\, A \leq [\alpha \mapsto \mu\alpha.\, C]\, B$.

Proof. Note that premise (2) can be obtained by inversion of premise (3). We explicitly show it here just for convenience. The whole proof follows a similar structure to Lemma 20: we proceed by induction on $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$.

- Case rule SA-VAR: In such case $A = B = \beta$. If $\beta \neq \alpha$, we simply achieve the goal.

  Otherwise,

  - ⋆ Goal (1): We want to prove $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\ C \leq \mu\alpha.\ D$, which is actually premise (3).
  - ⋆ Goal (2): From the condition of goal (2), we have $\Gamma_1, \alpha, \Gamma_2, \vdash_a D \leq C$, which is the inverse of premise (2). Thus, we get $C = D$ by Lemma 24. Goal (2) is proven by reflexivity.

- Case rule SA-ARROW: In such case $A = A_1 \to A_2$ and $B = B_1 \to B_2$.

  - ⋆ Goal (1):
    - We need to prove $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ C]\ (A_1 \to A_2) \leq [\alpha \mapsto \mu\alpha.\ D]\ (B_1 \to B_2)$, which can be rewritten as $\Gamma_1, \Gamma_2 \vdash_a ([\alpha \mapsto \mu\alpha.\ C]\ A_1) \to ([\alpha \mapsto \mu\alpha.\ C]\ A_2) \leq ([\alpha \mapsto \mu\alpha.\ D]\ B_1) \to ([\alpha \mapsto \mu\alpha.\ D]\ B_2)$.
    - By construction, we need to prove $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ D]\ B_1 \leq [\alpha \mapsto \mu\alpha.\ C]\ A_1$ and $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ C]\ A_2 \leq [\alpha \mapsto \mu\alpha.\ D]\ B_2$.
    - The former one can be proved by using induction hypothesis arising from goal (2), while the latter one can be proved by using induction hypothesis arising from goal (1).
  - ⋆ Goal (2):
    - We need to prove $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ D]\ (A_1 \to A_2) \leq [\alpha \mapsto \mu\alpha.\ C]\ (B_1 \to B_2)$, which can be rewritten as $\Gamma_1, \Gamma_2 \vdash_a ([\alpha \mapsto \mu\alpha.\ D]\ A_1) \to ([\alpha \mapsto \mu\alpha.\ D]\ A_2) \leq ([\alpha \mapsto \mu\alpha.\ C]\ B_1) \to ([\alpha \mapsto \mu\alpha.\ C]\ B_2)$.
    - By construction, we need to prove $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ C]\ B_1 \leq [\alpha \mapsto \mu\alpha.\ D]\ A_1$ and $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ D]\ A_2 \leq [\alpha \mapsto \mu\alpha.\ C]\ B_2$.
    - The former one can be proved by using induction hypothesis arising from goal (1), while the latter one can be proved by using induction hypothesis arising from goal (2).

- Case rule SA-REC: Now we assume $A = \mu\alpha'.\ A'$ and $B = \mu\alpha'.\ B'$. Since in such case, we do not need to consider the contravariance, we will just show how to prove goal (1). Goal (2) can be proven with the same approach.

  - The goal now is $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ C]\ \mu\alpha'.\ A' \leq [\alpha \mapsto \mu\alpha.\ D]\ \mu\alpha'.\ B'$, which can be rewritten as $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha'.\ [\alpha \mapsto \mu\alpha.\ C]\ A' \leq \mu\alpha'.\ [\alpha \mapsto \mu\alpha.\ D]\ B'$.

- The condition arising from the goal (1) becomes $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto C]\ \mu\alpha'.\ A' \leq [\alpha \mapsto D]\ \mu\alpha'.\ B'$, which can be rewritten as $\Gamma_1, \alpha, \Gamma_2 \vdash_a \mu\alpha'.\ [\alpha \mapsto C]\ A' \leq \mu\alpha'.\ [\alpha \mapsto D]\ B'$.
- Do inversion on this condition and reorder the context and substitution, we get two new conditions: $\Gamma_1, \alpha, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto C]\ A' \leq [\alpha \mapsto D]\ B'$ and $\Gamma_1, \alpha, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto C][\alpha' \mapsto A']\ A' \leq [\alpha \mapsto D][\alpha' \mapsto B']\ B'$.
- Because of the double unfolding rule, we will have two induction hypotheses, which are
  - ⋆ I.H.(1), which comes from 1-time unfolding : $\Gamma_1, \alpha, \Gamma_2, \vdash_a C \leq D \Rightarrow \Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\ C \leq \mu\alpha.\ D \Rightarrow \Gamma_1, \alpha, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto C]\ A' \leq [\alpha \mapsto D]\ B' \Rightarrow \Gamma_1, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto \mu\alpha.\ C]\ A' \leq [\alpha \mapsto \mu\alpha.\ D]\ B'$.
  - ⋆ I.H.(2), which comes from 2-times unfolding : $\Gamma_1, \alpha, \Gamma_2, \vdash_a C \leq D \Rightarrow \Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\ C \leq \mu\alpha.\ D \Rightarrow \Gamma_1, \alpha, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto C][\alpha' \mapsto A']\ A' \leq [\alpha \mapsto D][\alpha' \mapsto B']\ B' \Rightarrow \Gamma_1, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto \mu\alpha.\ C][\alpha' \mapsto A']\ A' \leq [\alpha \mapsto \mu\alpha.\ D][\alpha' \mapsto B']\ B'$.
- Apply construction on the goal, we obtain two sub-goals: $\Gamma_1, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto \mu\alpha.\ C]\ A' \leq [\alpha \mapsto \mu\alpha.\ D]\ B'$ and $\Gamma_1, \Gamma_2, \alpha' \vdash_a [\alpha' \mapsto A'][\alpha \mapsto \mu\alpha.\ C]\ A' \leq [\alpha' \mapsto B'][\alpha \mapsto \mu\alpha.\ D]\ B'$.
- For the former one, we apply the I.H.(1). As for the latter one, after rewriting the goal to $\Gamma_1, \Gamma_2, \alpha' \vdash_a [\alpha \mapsto \mu\alpha.\ C][\alpha' \mapsto A']\ A' \leq [\alpha \mapsto \mu\alpha.\ D][\alpha' \mapsto B']\ B'$, we apply the I.H.(2).

□

Like Lemma 20, in Lemma 25 the two conclusions are basically reflecting two lemmas: one for covariant uses (when A is a part of C and B is a part of D), and another for contravariant uses (when A is a part of D and B is a part of C). By letting $C := A$, $D := B$, we easily obtain:

Lemma 26. Unfolding Lemma.

$$\text{If } \Gamma \vdash_a \mu\alpha.\ A \leq \mu\alpha.\ B \text{ then } \Gamma \vdash_a [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B.$$

4.5 Nominal Unfolding

In this section, we will describe the nominal unfolding rule, which is another algorithmic variant equivalent to declarative subtyping. Compared with the double unfolding rules, nominal unfoldings have better efficiency (since only one premise is needed), while eliminating spurious subtyping derivations that arise with double unfoldings (see example in Chapter 4.1). As in the previous section, we present a variant of the calculus in Chapter 3, but this time using the nominal unfolding rules instead.

4.5.1 Syntax and well-formedness

The syntax of contexts for this calculus is the same as Chapter 3.2. For the syntax of types, based on the syntax in Chapter 3.2, we extend it with *labelled types* A^{α}. Labelled types can be viewed as a simple form of nominal types. They are essentially a pair that contains a name (or type variable) α and a type.

The well-formedness $\Gamma \vdash A$ is also defined as Chapter 3.2, but for recursive types and labelled types (the top of Figure 4.2). To get a better induction hypothesis, we slightly modify the form of well-formed recursive types, as rule WFT-NOMINAL shows. As before, rule WFT-NOMINAL is proven to be equivalent to rule WFT-REC. The first premise $\Gamma, \alpha \vdash A$ might appear redundant at first glance, but it is indeed necessary, because from the second premise $\Gamma, \alpha \vdash [\alpha \mapsto A^{\alpha}]\, A$, we cannot derive $\Gamma, \alpha \vdash A$, which is the insurance with respect to the correctness of substitution during the proof. Meanwhile, since we introduce labelled types, as rule WFT-FLABEL shows, a labelled type is well-formed if its inner type is well-formed.

4.5.2 Subtyping

The bottom of Figure 4.2 shows the definition of subtyping with the nominal unfolding rule. We denote subtyping for nominal unfoldings as $\Gamma \vdash_n A \leq B$. Rules SN-NAT, SN-TOP, SN-VAR, and SN-ARROW are the same as the corresponding double unfolding subtyping rules. Rule SN-RCD is new, stating that a labelled type is a subtype of another labelled type if the two types are labelled with the same name and $A \leq B$.

Rule SN-REC, the nominal unfolding rule, is the most interesting one. This rule follows an idea quite similar to the double unfolding rule. The body of the recursive type is unfolded twice. However, for the innermost unfolding, the type that we substitute is not the type of the body directly. Instead, we use a labelled type, where the label has a fresh name, and the type that is labelled is the body of the recursive type. In other words, instead of using the double unfolding $[\alpha \mapsto A]\, A$ we use $[\alpha \mapsto A^{\alpha}]\, A$. The label is crucial to avoid spurious subtyping derivations, and it is also the reason why in the nominal unfolding formulation we do not need to check the subtyping of single unfoldings as well. In the double unfolding rule, there is an extra premise that checks the single unfolding and prevents certain cases of spurious subtyping. The absence of this extra premise also makes some of the metatheory simpler.

4.5.3 Basic properties

All the proofs about reflexivity, transitivity and unfolding lemma for nominal unfoldings are almost the same as double unfoldings, since both subtyping rules are based on two times finite unfoldings. We list all the theorems here and skip the details (the reader can consult our mechanized proofs for full details).

Theorem 27. Reflexivity.

$$\text{If } \vdash \Gamma \text{ and } \Gamma \vdash A \text{ then } \Gamma \vdash_n A \leq A.$$

$\boxed{\Gamma \vdash A}$ *(Well-Formed Type, Selected Rules)*

$$\textsc{wft-flabel}\ \frac{\Gamma \vdash A}{\Gamma \vdash A^{\alpha}} \qquad \textsc{wft-nominal}\ \frac{\Gamma, \alpha \vdash A \qquad \Gamma, \alpha \vdash [\alpha \mapsto A^{\alpha}]\ A}{\Gamma \vdash \mu\alpha.\ A}$$

$\boxed{\Gamma \vdash_n A \leq B}$ *(Nominal Subtyping)*

$$\textsc{sn-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash_n \mathsf{nat} \leq \mathsf{nat}} \qquad \textsc{sn-top}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash_n A \leq \top} \qquad \textsc{sn-var}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \alpha}{\Gamma \vdash_n \alpha \leq \alpha}$$

$$\textsc{sn-arrow}\ \frac{\Gamma \vdash_n B_1 \leq A_1 \qquad \Gamma \vdash_n A_2 \leq B_2}{\Gamma \vdash_n A_1 \to A_2 \leq B_1 \to B_2} \qquad \textsc{sn-rcd}\ \frac{\Gamma \vdash_n A \leq B}{\Gamma \vdash_n A^{\alpha} \leq B^{\alpha}}$$

$$\textsc{sn-rec}\ \frac{\Gamma, \alpha \vdash_n [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B}{\Gamma \vdash_n \mu\alpha.\ A \leq \mu\alpha.\ B}$$

Figure 4.2: Well-formedness and subtyping rules for nominal unfoldings.

Theorem 28. Transitivity.

$$\text{If } \Gamma \vdash_n A \leq B \text{ and } \Gamma \vdash_n B \leq C \text{ then } \Gamma \vdash_n A \leq C.$$

Lemma 29. Unfolding Lemma.

$$\text{If } \Gamma \vdash_n \mu\alpha.\ A \leq \mu\alpha.\ B \text{ then } \Gamma \vdash_n [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B.$$

Another important property is that, from the nominal unfolding rules, we can derive 1-time finite unfoldings. This lemma is important to show that nominal unfoldings subsume the double unfolding rule:

Lemma 30. If $\Gamma, \alpha \vdash_n [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B$ then $\Gamma, \alpha \vdash_n A \leq B$.

4.5.4 Equivalence between nominal unfoldings and double unfoldings

The subtyping relation presented in Chapter 4.5 is equivalent to a subtyping relation that uses the double unfolding rules for recursive types. This equivalence is not surprising, since nominal unfoldings are essentially the double unfolding rule with an extra label and without the 1-time finite unfolding premise. Lemma 30 and some other similar auxiliary lemmas are used to formulate the equivalence between the two encodings. The most interesting aspect of the equivalence proof is that we need to translate types for the nominal unfolding formulation into types of the double-unfolding formulation. Such a translation is necessary because nominal unfoldings require labelled types, which do not exist in the double unfolding formulation. Thus, the translation simply erases the labels.

Definition 31. The *erase* ($\searrow$) function is defined as:

$$\begin{array}{rcl} \mathrm{nat}\searrow & = & \mathrm{nat} \\ \top\searrow & = & \top \\ \alpha\searrow & = & \alpha \\ (A \to B)\searrow & = & A\searrow \to B\searrow \\ (\mu\alpha.\ A)\searrow & = & \mu\alpha.\ A\searrow \\ A^{\alpha}\searrow & = & A\searrow \end{array}$$

With the erasure function we can conclude that our nominal unfoldings are equivalent to double unfoldings with the following two lemmas:

Theorem 32. If $\Gamma \vdash_n A \leq B$ then $\Gamma \vdash_a A\searrow \leq B\searrow$.

Theorem 33. If $\Gamma \vdash_a A \leq B$ then $\Gamma \vdash_n A \leq B$.

For Theorem 32 we wish to show that all valid subtyping statements using nominal unfoldings are also valid under the double unfolding formulation. To show this result we have to apply the erasure function to the types, since the types in the nominal unfolding formulation may contain labels. For Theorem 33 no erasure function is necessary since the types in the double unfolding formulation are a subset of those in the nominal unfolding formulation. Thus, they can be directly mapped. As a consequence of the two theorems above, our nominal unfoldings are also sound and complete with respect to our specification using finite unfoldings.

Corollary 34. If $\Gamma \vdash_n A \leq B$ then $\Gamma \vdash A\searrow \leq B\searrow$.

Corollary 35. If $\Gamma \vdash A \leq B$ then $\Gamma \vdash_n A \leq B$.

4.6 Decidability

Our subtyping rules are decidable. We have already proved the equivalence between the rules employing nominal, double and finite unfoldings. Since both the nominal and the double unfolding rules are syntax directed, they provide a useful foundation to prove decidability. We have proved decidability based on our nominal rule and a measure that is based on the depth of the unfolded tree. A similar proof should be possible using the double unfolding rule, except that with the double unfolding rule there is some extra work because of the extra 1-time finite unfolding premise.

Our subtyping rules are based on substitution, which can increase the size of types after an unfolding. Therefore, a straightforward induction on the size of types will not work. A first idea may be doing induction lexicographically on a pair with the number of nesting of recursive binders, and the size of types. The logic is that, after a nominal unfolding, the recursive binder that we are going to unfold will not reappear again. However, this does not quite work because the bodies of recursive types can contain other recursive types and the substitutions may introduce new copies of those recursive types. Thus, the subtyping rule for recursive types does not necessarily reduce the number of recursive binders. Consider, for instance, the following example:

$$\mu\alpha.\ \mu\beta.\ \alpha \to \beta$$

After the nominal unfolding and α-conversion, the type will become:

$$\mu\beta.\ (\mu\beta'.\ \alpha \to \beta')^{\alpha} \to \beta$$

which does not decrease the number of recursive binders. Nevertheless, if we continue to process the types using nominal unfolding, we will finally reach a type without any recursive binders. After a few more steps in the subtyping derivation, we obtain:

$$(\mu\beta'.\ \alpha \to \beta')^{\alpha} \to ((\mu\beta'.\ \alpha \to \beta')^{\alpha} \to \beta)^{\beta}$$

and the inner recursive types $\mu\beta'.\ \alpha \to \beta'$ no longer contain recursive types in their bodies, and we will finally obtain types free of recursive types after another round of nominal unfolding.

4.6.1 Measure based on the depth of the unfolded tree

To provide a measure that decreases at every nominal unfolding, we define a function based on the depth of the expanded tree of a type. This function essentially simulates the unfolding process of the tree using nominal unfoldings and allow us to obtain an (over-)approximation of the depth of the (fully) unfolded tree.

Definition 36. The *height* of a type A in a context Ψ ($\Psi := \cdot \mid \Psi, \alpha \mapsto i$, where i represents a natural number), written $height_{\Psi}(A)$, is defined as follows:

$$\begin{array}{lcll}
height_{\Psi}(\mathrm{nat}) & = & 0 & \\
height_{\Psi}(\top) & = & 0 & \\
height_{\Psi}(A_1 \to A_2) & = & max(height_{\Psi}(A_1),\ height_{\Psi}(A_2)) + 1 & \\
height_{\Psi}(A^{\alpha}) & = & height_{\Psi}(A) + 1 & \\
height_{\Psi}(\alpha) & = & \Psi(\alpha) & (\alpha \in \Psi) \\
height_{\Psi}(\alpha) & = & 0 & (\alpha \notin \Psi) \\
height_{\Psi}(\mu\alpha.\ A) & = & \text{let } i = height_{\Psi,\ \alpha \mapsto 0}(A) \text{ in } height_{\Psi,\ \alpha \mapsto i+1}(A) + 1 &
\end{array}$$

The two interesting cases in the *height* function are for recursive variables and recursive types. For a recursive variable, if it can be found in the context, we retrieve the corresponding height associated with the recursive variable from the context, whereas we return 0 if α is not in the context. Note that, when performing subtyping on two closed types (which is always the case in the subsumption rule) the latter case never happens. However, to make *height* total we have to consider this case too, and therefore our height function applies even to types which are not well-formed. For a recursive type, we firstly compute the height of its body by assuming that the height of its binder is 0. In other words i is the height of the 1-time finite unfolding. Then we compute the height of the body again, but this time assuming that the height of its binder is $i + 1$ (i.e. the size of the 1-time unfolding plus 1). This basically computes the overall height of the nominal unfolding. Since we compute the height two times for a recursive type, our *height* function is convex: its second derivative with respect to the number of recursive types is non-negative thus a linear over-approximation is impossible.

Finally, the measure of a type A is defined as $height(A)$, which is the height of expanded tree when the context Ψ is empty.

4.6.2 The decidability proof

With the new measure, now we can prove the decidability lemma. For a non-recursive type, it is obvious that the height of a conclusion from any inputs is strictly greater than the height of its any premises. For a recursive type, the measure will decrease by 1 after a nominal unfolding. In other words, what we want to show is

$$height(\mu\alpha.\ A) - 1 = height_{\alpha \mapsto height_{\alpha \mapsto 0}(A)+1}(A) = height([\alpha \mapsto A^{\alpha}]\ A).$$

Firstly, it is easy to observe that $height_{\alpha \mapsto 0}(A) = height(A)$, because for a variable, it is either found at the context, which is $\Psi(\alpha) = 0$, or not found at the context, which will return 0. Then, when we try to compute $height([\alpha \mapsto A^{\alpha}]\ A)$, since α is substituted by A^{α} and α is not in the context, the formula can be rewritten as $height_{\alpha \mapsto height(A^{\alpha})}(A)$, in which we do not try to proceed with the substitution, but just return the result from the context. We can continue to rewrite this formula as $height_{\alpha \mapsto height(A)+1}(A)$. Through $height_{\alpha \mapsto 0}(A) = height(A)$, the formula is equal to $height_{\alpha \mapsto height_{\alpha \mapsto 0}(A)+1}(A)$. Therefore, we have proven our proposition. Next we can prove that this measure suffices to show the termination of subtyping with nominal unfoldings:

Lemma 37. If $max(height(A),\ height(B)) \leq k$, then $\Gamma \vdash_n A \leq B$ or not $\Gamma \vdash_n A \leq B$.

Proof. Do induction on k, A and B, respectively. □

Let $k = max(height(A),\ height(B))$, we obtain:

Theorem 38. Termination:

For any inputs Γ, A and B, we either have $\Gamma \vdash_n A \leq B$ or not $\Gamma \vdash_n A \leq B$.

Finally, from termination and the soundness and completeness of subtyping based on nominal unfoldings with respect to subtyping based on finite unfoldings we can conclude that our specification of iso-recursive subtyping is decidable.

Corollary 39. Decidability: Our specification for iso-recursive subtyping is decidable.

As a final note, although we use height as measure here, we can also use weight as measure, which is no essentially difference. In other words, instead of a maximum function for function case, an addition function is also feasible. In later chapters, we will show an alternative by employing weight measure, demonstrating that our technique is very standard.

4.7 Discussion

As we shall see, both the double and the nominal unfolding rules are easy to work with in terms of proofs and metatheory, and the nominal unfolding rules can even simplify some proofs due

to the single premise. The double unfolding rule is directly inspired by the finite unfolding specification. The nominal unfolding rule additionally employs the idea of tracking the recursive type variable as a label to avoid spurious subtyping that arises from double unfoldings. Therefore, it can avoid the extra 1-time finite unfolding check. In the nominal unfolding rule it is interesting to observe that the names of recursive type variables play an important role, just as in the Amber rules. However, in the Amber rules, we use distinct type variable names and track the subtyping relation between those variables. In the nominal unfolding rule we use the same type variable name, which is sufficient to identify types that originate from the double unfolding substitutions. Therefore, spurious subtyping when using only double unfoldings can be avoided in the nominal unfolding rule. We will have a more discussion on spurious subtyping problem in Chapter 6.

4.7.1 Some final implementation considerations

The double unfolding and nominal unfolding rules are primarily designed with the goal of leading to a simple metatheory and proofs. Both rules employ substitutions which, if used directly in an implementation, have significant performance penalties. To avoid substitutions one possibility would be to adopt *explicit substitutions* [2], which are a standard solution to avoid the performance penalties associated with substitutions. Another possibility would be to adopt some ideas in the implementation approach proposed by Ligatti et al. [84]. Although Ligatti et al. [84]'s rules have different expressive power compared to the Amber rules and our rules, they also employ substitutions. They present an optimized $O(mn)$ algorithm that avoids the use of substitutions, and we believe that it should be possible to adopt some of those ideas to implement double/nominal unfoldings. Finally, a simple optimization for both double and nominal unfoldings is to avoid substitutions in *positive* positions. As Chapter 2.1.1 discusses for covariant subtyping using $\Gamma \vdash A \leq B$ in the premise of the recursive subtyping rule is sound. Thus, we should not need to substitute recursive type variables that are found in positive positions, which avoids extra subtype checks of the substituted types. In other words, we could have the variant (here for nominal unfoldings):

$$\frac{\Gamma, \alpha \vdash [\alpha \mapsto A^{\alpha}]^{+}\ A \leq [\alpha \mapsto B^{\alpha}]^{+}\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}\ \text{S-fnominal+}$$

The idea is to employ a polarized form of substitution $[\alpha \mapsto A]^{m}\ B$, which is parametrized by a positive $(+)$ or negative $(-)$ mode m. This form of substitution would only perform substitutions at negative occurrences of type variables. Thus, the special case of covariant subtyping would behave equivalently to the rule presented in Chapter 2.1.1. We leave the development and proof of correctness for an efficient algorithm for future work.

4.8 Mechanized Proofs

The folder *coq_theory* includes all the Coq proofs about algorithmic iso-recursive subtyping in this chapter, namely double unfolding rules and nominal unfolding rules.

Table 4.1: Paper-to-proofs correspondence guide in Chapter 4.

Definition	File	Name in Coq	Notation
Well-formed Type (Definition 15)	Rules.v	WF E A	$\Gamma \vdash A$
Well-formed Type (Figure 4.2)	NominalUnfolding.v	Nominal.WFS E A	$\Gamma \vdash A$
Double Unfolding Rule (Figure 4.1)	Rules.v	sub E A B	$\Gamma \vdash_a A \leq B$
Nominal Unfolding Rule (Figure 4.2)	NominalUnfolding.v	Nominal.Sub E A B	$\Gamma \vdash_n A \leq B$

Table 4.2: Descriptions for the proof scripts in Chapter 4.

Theorems	Description	Files	Name in Coq
Theorem 17	Reflexivity (Double)	FiniteUnfolding.v	refl_algo
Theorem 18	Transitivity (Double)	FiniteUnfolding.v	trans_algo
Theorem 19	Completeness (Double)	FiniteUnfolding.v	completeness
Theorem 23	Soundness (Double)	DoubleUnfolding.v	soundness
Lemma 26	Unfolding Lemma (Double)	DoubleUnfolding.v	unfolding_lemma_version2
Theorem 27	Reflexivity (Nominal)	NominalUnfolding.v	Nominal.sub_refl
Theorem 28	Transitivity (Nominal)	NominalUnfolding.v	Nominal.Transitivity
Lemma 29	Unfolding Lemma (Nominal)	NominalUnfolding.v	Nominal.unfolding_lemma
Theorem 32	Nominal to Double	NominalUnfolding.v	nominal_to_double
Theorem 33	Double to Nominal	NominalUnfolding.v	double_to_nominal
Corollary 34	Soundness (Nominal)	NominalUnfolding.v	nominal_to_finite
Corollary 35	Completeness (Nominal)	NominalUnfolding.v	finite_to_nominal
Theorem 38	Decidability	Decidability.v	decidability

4.8.1 Definitions

All the definitions in the Chapter 4 can be found in files *Rules.v* and *NominalUnfolding.v*. Table 4.1 shows the correspondence of definitions between the paper and the Coq artifacts. The file *Rules.v* contains the definitions for the double unfoldings. It has definitions of well-formedness, subtyping, typing, and reduction. The file *NominalUnfolding.v* contains the definitions involving nominal unfoldings.

Note that, in the rule rule SN-REC, we have reused the recursive variable name for the label α. However, the labels and type variable names should be considered distinct. Labels α should be the same in both substitutions, and distinct from any other labels and bound variables used elsewhere.

4.8.2 Lemmas and theorems

Table 4.2 shows the descriptions for all the proof scripts in Chapter 4. For succinctness, we briefly describe the important lemmas and theorems.

Lemma 25 in paper:
If
1. $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$;
2. $\Gamma_1, \alpha, \Gamma_2 \vdash_a C \leq D$;
3. $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\, C \leq \mu\alpha.\, D$;

then
1. $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto C]A \leq [\alpha \mapsto D]B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, C]A \leq [\alpha \mapsto \mu\alpha.\, D]B$ and
2. $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto D]A \leq [\alpha \mapsto C]B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, D]A \leq [\alpha \mapsto \mu\alpha.\, C]B$.

Lemma 25 in Coq:

If
1. $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$;
2. $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\, C \leq \mu\alpha.\, D$;
3. $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto C \oplus_m D]A \leq [\alpha \mapsto D \oplus_m C]B$

then
$\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\, C \oplus_m D]A \leq [\alpha \mapsto \mu\alpha.\, D \oplus_m C]B$

Figure 4.3: Comparison between paper and Coq statements for Lemma 25.

An important difference between some of the lemma statements in the paper and the Coq proofs is that we make more use of modes in Coq. This change is done for readability purposes. In particular, all variants of the unfolding lemma in the paper are presented without modes in the paper. Figure 4.3 illustrates the difference between the formulations with and without modes for the unfolding lemma (note that the premise (2) is redundant since it is the inversion of the premise (3), thus in the Coq code we drop this premise while in the paper presentation we keep it for readability). Our Coq formalization uses some meta-functions on modes instead to formalize the same result. Using meta-functions on modes (Definition 40), the same lemma would look like the right part of Figure 4.3.

Definition 40. Mode selector.

$$C \oplus_+ D = C \quad C \oplus_- D = D$$

In the Coq proof, we also defined some special notations for definitions representing n-times finite unfolding, and for the meta-functions on modes. Those definitions can be found in the file *Rules.v*.

Another important difference is in the decidability proof. Unlike the paper proof, where in the context we store the variable names as keys, in the Coq proof we employ De Bruijn indices to represent all recursive variables stored in the context.

Chapter 5

Weakly Positive Subtyping

In addition to the Amber rules and the finite and the rules based on double unfoldings, we will give another equivalent formulation of subtyping based on a weakly positive restriction of recursive variables. This variant captures precisely a folklore observation that the Amber rules express two situations where a recursive variable can be a subtype of another type: positive subtyping and reflexivity. This variant, presented as part of Chapter 5, is used as an intermediate step to prove the equivalence between the Amber rules and a formulation using double unfolding.

5.1 Subtyping Based on a Weakly Positive Restriction

In this chapter we will show a variant of the Amber rules that is equivalent, in terms of expressive power, to our new formulation of subtyping based on finite unfoldings. We prove the equivalence via soundness and completeness theorems between the two formulations of subtyping. The soundness lemma implies that if two types are subtypes under the Amber rules, they are subtypes under our new formulation. The completeness lemma implies that if two types are subtypes under our new formulation, they are subtypes under the Amber rules. With both lemmas we can conclude that our formulation and the Amber rules have the same expressiveness.

To prove the soundness and completeness with respect to a formulation based on finite unfoldings we create an intermediate subtyping relation to make the proof easier. This intermediate relation, presented in Figure 5.1, is equivalent to the Amber rules in Figure 5.2. The key idea in this relation is to have a rule for recursive types (rule PosRes-rec), which only accepts weakly positive subtyping. This formulation is inspired by the existing positive formulation of subtyping for recursive types [5, 10, 12], but it is more general.

In essence, what we mean by weakly positive subtyping is that we can never find a contravariant subderivation $\alpha \leq \alpha$, where α is a recursive type variable, for *non-equal* recursive types. For instance this excludes $\mu\alpha.\alpha \to \mathsf{nat} \leq \mu\alpha.\alpha \to \top$, since here α is used contravariantly, and $\alpha \leq \alpha$ would appear as a subderivation. Notice, however, that weakly positive subtyping still allows subtyping of recursive types with negative occurrences of the recursive type variable in two cases:

$\boxed{\alpha \in_m A \leq B}$ *(Weakly Positive Restriction)*

$$\textsc{Posvar-nat}\ \frac{}{\alpha \in_m \mathsf{nat} \leq \mathsf{nat}} \qquad \textsc{Posvar-topl}\ \frac{}{\alpha \in_m A \leq \top} \qquad \textsc{Posvar-topr}\ \frac{}{\alpha \in_m \top \leq A}$$

$$\textsc{Posvar-arrow}\ \frac{\alpha \in_{\overline{m}} B_1 \leq A_1 \qquad \alpha \in_m A_2 \leq B_2}{\alpha \in_m A_1 \to A_2 \leq B_1 \to B_2} \qquad \textsc{Posvar-varx}\ \frac{}{\alpha \in_+ \alpha \leq \alpha} \qquad \textsc{Posvar-vary}\ \frac{\alpha \neq \beta}{\alpha \in_m \beta \leq \beta}$$

$$\textsc{Posvar-recself}\ \frac{\beta \notin fv(A)}{\beta \in_m \mu\alpha.\, A \leq \mu\alpha.\, A} \qquad \textsc{Posvar-rec}\ \frac{\beta \in_m A \leq B \qquad \alpha \in_+ A \leq B \qquad \alpha \neq \beta}{\beta \in_m \mu\alpha.\, A \leq \mu\alpha.\, B}$$

$\boxed{\Gamma \vdash A \leq_+ B}$ *(Weakly Positive Subtyping)*

$$\textsc{PosRes-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nat} \leq_+ \mathsf{nat}} \qquad \textsc{PosRes-top}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash A \leq_+ \top} \qquad \textsc{PosRes-var}\ \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash \alpha \leq_+ \alpha}$$

$$\textsc{PosRes-arrow}\ \frac{\Gamma \vdash B_1 \leq_+ A_1 \qquad \Gamma \vdash A_2 \leq_+ B_2}{\Gamma \vdash A_1 \to A_2 \leq_+ B_1 \to B_2} \qquad \textsc{PosRes-rec}\ \frac{\Gamma, \alpha \vdash A \leq_+ B \qquad \alpha \in_+ A \leq B}{\Gamma \vdash \mu\alpha.\, A \leq_+ \mu\alpha.\, B}$$

$$\textsc{PosRes-self}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \mu\alpha.\, A}{\Gamma \vdash \mu\alpha.\, A \leq_+ \mu\alpha.\, A}$$

Figure 5.1: Weakly positive subtyping rules.

- **Equal types:** If the recursive types are equal, then weakly positive subtyping still considers the two types to be subtypes. For instance $\mu\alpha.\alpha \to \mathsf{nat} \leq \mu\alpha.\alpha \to \mathsf{nat}$, is a valid subtyping statement.
- **The recursive type variable is a subtype of $\top$:** If a recursive type variable appears negatively, but the only (negative) subderivations are of the form $\alpha \leq \top$, then that is allowed in weakly positive subtyping. For instance $\mu\alpha.\top \to \alpha \leq \mu\alpha.\alpha \to \alpha$ is a valid weakly positive subtyping statement.

These two exceptions are why we use the term "weakly" to characterize such formulation of subtyping. In contrast, existing formulations of positive subtyping, such as that described in Chapter 2.1.1 or originally described by Amadio and Cardelli [5] do not make such exceptions and would reject the subtyping statements that we have described above.

5.1.1 Well-formedness and weakly positive relation

Well-formed types are the same as in Figure 3.3. To examine whether a type variable occurs positively in a subtyping relation, we define a weakly positive restriction relation $\alpha \in_m A \leq B$ at the top of Figure 5.1. Here, $\alpha \in_m A \leq B$ means that: type variable α occurs in the derivation

$A \leq B$ with a mode m, where a mode m is either positive (+) or negative (-)[1]. This relation checks that every instance of $\alpha \leq \alpha$ in the proof derivation of $A \leq B$ is found in a positive position inside the proof (rule Pos-varx). Moreover, for every subderivation of $A \leq B$ with shape $\mu\beta.\ A' \leq \mu\beta.\ B'$ either 1) $A' = B'$ and α is not free in A' (rule Posvar-recself), or 2) $\beta \in_+ A' \leq B'$ (rule Posvar-rec).

For example, $\alpha \in_+ \top \to \alpha \leq \alpha \to \alpha$ holds, since the only instance of $\alpha \leq \alpha$ occurs positively and there are no recursive types inside, so the second condition does not apply. To see the need for the second condition, consider:

$$\beta \in_+ \mu\alpha.\ \alpha \to \beta \leq \mu\alpha.\ \alpha \to \beta$$

which might seem to hold according to the syntax, since β appears only in positive positions. However, it is rejected by both rule Posvar-rec and rule Posvar-recself. Rule Posvar-rec requires that α also appears positively in subderivations, which does not hold in this example. The reason we pose such restriction is because, for instance, unfolding both types results in the following judgment

$$\beta \in_+ (\mu\alpha.\ \alpha \to \beta) \to \beta \leq (\mu\alpha.\ \alpha \to \beta) \to \beta$$

where a negative occurrence of $\beta \leq \beta$ would appear in a subderivation. A similar issue happens whenever $\alpha \leq \alpha$ appears negatively and the recursive types are not equal to each other.

There are also some noteworthy points in the other rules for the weakly positive restriction relation. In rule Posvar-arrow, for the contravariant types, we switch their mode by a flip operation $\overline{m}$: $\overline{+} = -$ and $\overline{-} = +$. Rule Posvar-vary states that if α is not equal to β, we do not care what the mode for β is. Rule Posvar-topr, at first glance, looks suspicious, since it seems to indicate that $\top \leq A$ is valid. In this rule the choice of notation for the relation, using $\leq$, may be a little misleading. Although normally we follow the derivation of the subtyping relation, the mode is determined by the position and not by whether the two types are subtypes. The addition of rule Posvar-topr is not harmful: the relation is always accompanied by weakly positive subtyping derivations, and $\top \leq A$ never occurs in such subtyping derivations. The reason to include rule Posvar-topr is that we wish that our weakly positive restriction relation is symmetric: if $\alpha \in_m A \leq B$ then $\alpha \in_m B \leq A$. This symmetry property is important for the proof of Lemma 50.

5.1.2 Subtyping

Most subtyping rules are identical to those of the Amber rules, and the only differences are rule PosRes-var, rule PosRes-rec and rule PosRes-self. The rule PosRes-var is similar to our formulations, checking whether two variables are same. The latter two rules state that: 1) two recursive types are subtypes if they are equal (rule PosRes-self); or 2) the recursive variable satisfies the weakly positive restriction and the two bodies are subtypes (rule PosRes-rec).

[1]Note that $\in_m$ is just part of the syntax of the relation, rather than a separate operator.

5.1.3 Basic properties

The reflexivity is straightforward since we have explicit reflexivity built-in for recursive types.

Theorem 41. Reflexivity.

$$\text{If } \vdash \Gamma \text{ and } \Gamma \vdash A \text{ then } \Gamma \vdash A \leq_+ A.$$

As for transitivity, because we have the weakly positive restriction for recursive subtyping, the proof is a bit complex. We need to prove an auxiliary lemma in advance:

Lemma 42. If

1. $\Gamma \vdash A \leq_+ B$;
2. $\Gamma \vdash B \leq_+ C$;
3. $\alpha \in_m A \leq B$;
4. $\alpha \in_m B \leq C$,

then $\Gamma \vdash A \leq_+ C$ and $\alpha \in_m A \leq C$.

Proof. Induction on $\Gamma \vdash B$. □

Then we can have the transitivity theorem.

Theorem 43. Transitivity.

$$\text{If } \Gamma \vdash A \leq_+ B \text{ and } \Gamma \vdash B \leq_+ C \text{ then } \Gamma \vdash A \leq_+ C.$$

Proof. Induction on $\Gamma \vdash B$. For the recursive case, apply lemma 42, we have all premises. □

Finally, it is also possible to prove the unfolding lemma for weakly positive subtyping:

Lemma 44. Unfolding Lemma.

$$\text{If } \Gamma \vdash \mu\alpha.\ A \leq_+ \mu\alpha.\ B \text{ then } \Gamma \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq_+ [\alpha \mapsto \mu\alpha.\ B]\ B.$$

The proof employs similar techniques to those used for the soundness lemma (Lemma 52). We skip the details here.

5.2 The Formalization of Amber Rules

In the following sections, we are going to show all the equivalence theorems. However, there is still a challenge: how to define iso-recursive Amber rules formally. In the original Amber rules by Amadio and Cardelli [5] (Figure 2.1) there are no well-formedness constraints. Unfortunately, defining such well-formedness constraints is not entirely trivial. Furthermore,

for those interested in mechanical formalization using theorem provers (as we are), such details need to be spelled out clearly. Well-formedness usually plays an important role in the metatheory, since some proofs can be more easily proved by considering well-formed types and environments only. One typical property of subtyping that we may hope to have is the so-called regularity of subtyping:

$$\text{If } \Gamma \vdash A \leq B \text{ then } \vdash \Gamma \wedge \Gamma \vdash A \wedge \Gamma \vdash B.$$

which states that if a subtyping statement is valid then the context and types are well-formed. Regularity is typically used in many other proofs, such as the proof of transitivity in algorithmic formulations. Note that, in the Amber rules, the rule for recursive types uses two distinct type variables α and β in the recursive types. The use of such distinct type variables is a crucial feature of the Amber rules and is used to prevent subderivations of the form $\Gamma \vdash \beta \leq \alpha$, where Γ only contains $\alpha \leq \beta$ but not $\beta \leq \alpha$. Otherwise, if such subderivations would be accepted, type soundness would be broken.

With the Amber rules an intuitive idea is that the subtyping environment consists of a sequence of pairs of type variables $\overline{\alpha \leq \beta}$ and that the α's are in scope on the type at the left-side of the subtyping relation (A), while the β's are in scope in the type at the right-side of the subtyping relation (B). Sadly, this idea is not that simple to realise. Note that in the subtyping rule of function types (rule Amber-arrow), the input arguments are swapped, so without any changes in the environment the type variables in the types would go out-of scope, and this breaks the regularity lemma. Furthermore, trying to perhaps swap the variables in the environment to keep them in-scope changes the meaning of the environment ($\alpha \leq \beta$ becomes $\beta \leq \alpha$). Trying to ensure that the α's are only in scope in one side of the relation, while the β's are only in scope in the other side, turns out to be quite tricky. Therefore, to make progress, we propose a weaker restriction in this section: we allow both α's and β's to be in scope for both types. Thus, the following subtyping statement is valid with our variant of the Amber rules: $\alpha \leq \beta \vdash \alpha \to \beta \leq \top$. In other words, we accept some subtyping statements that one would perhaps expect to be ill-formed or rejected. That is, in the Amber rules, if we have $\alpha \leq \beta$ in Γ, we would not expect that α and β appear in the same type. Rather we would expect that the α appears in one of the types, and β in the other one. However, accepting such subtyping statements is not harmful: we can still prove the soundness and completeness of this variant with respect to our new formulation of subtyping.

5.2.1 Well-formed environment and types

In the Amber rules, the subtyping context stores pairs of distinct type variables. We use:

$$\Delta := \cdot \mid \Delta, \alpha \leq \beta$$

to denote the context for Amber rules. Figure 5.2 shows a set of standard Amber rules with a built-in reflexivity rule.

A well-formed environment ($\vdash \Delta$) requires that all pairs of variables ($\alpha \leq \beta$) in the environment Δ are distinct. Well-formed types are almost standard, except that both α and β are

$\boxed{\Delta \vdash A}$ *(Well-Formed Type of Amber Rules)*

$$\textsc{WFAmber-nat}\ \frac{\vdash \Delta}{\Delta \vdash \mathsf{nat}} \qquad \textsc{WFAmber-Top}\ \frac{\vdash \Delta}{\Delta \vdash \top} \qquad \textsc{WFAmber-varl}\ \frac{\vdash \Delta \qquad \alpha \leq \beta \in \Delta}{\Delta \vdash \alpha} \qquad \textsc{WFAmber-varr}\ \frac{\vdash \Delta \qquad \alpha \leq \beta \in \Delta}{\Delta \vdash \beta}$$

$$\textsc{WFAmber-arrow}\ \frac{\Delta \vdash A_1 \qquad \Delta \vdash A_2}{\Delta \vdash A_1 \to A_2} \qquad \textsc{WFAmber-rec}\ \frac{\Delta, \alpha \leq \beta \vdash A \qquad \beta \text{ is fresh}}{\Delta \vdash \mu\alpha.\, A}$$

$\boxed{\Delta \vdash_{amb} A \leq B}$ *(Amber Rules)*

$$\textsc{Amber-nat}\ \frac{\vdash \Delta}{\Delta \vdash_{amb} \mathsf{nat} \leq \mathsf{nat}} \qquad \textsc{Amber-top}\ \frac{\vdash \Delta \qquad \Delta \vdash A}{\Delta \vdash_{amb} A \leq \top} \qquad \textsc{Amber-arrow}\ \frac{\Delta \vdash_{amb} B_1 \leq A_1 \qquad \Delta \vdash_{amb} A_2 \leq B_2}{\Delta \vdash_{amb} A_1 \to A_2 \leq B_1 \to B_2}$$

$$\textsc{Amber-var}\ \frac{\vdash \Delta \qquad \alpha \leq \beta \in \Delta}{\Delta \vdash_{amb} \alpha \leq \beta} \qquad \textsc{Amber-rec}\ \frac{\Delta, \alpha \leq \beta \vdash_{amb} A \leq B}{\Delta \vdash_{amb} \mu\alpha.\, A \leq \mu\beta.\, B} \qquad \textsc{Amber-self}\ \frac{\vdash \Delta \qquad \Delta \vdash \mu\alpha.\, A}{\Delta \vdash_{amb} \mu\alpha.\, A \leq \mu\alpha.\, A}$$

Figure 5.2: A variant of the Amber rules, including well-formedness of types.

considered declared by a pair $(\alpha \leq \beta)$ in the context (rule WFAmber-varl and rule WFAmber-varr), and rule WFAmber-rec introduces a pair of fresh variables into the context, although the second variable is never used. Rule WFAmber-rec simply mimics the left-hand side derivation of rule Amber-rec of the Amber subtyping relation, as we shall see next. With our definition of well-formed types regularity is easy to obtain:

Lemma 45. Regularity: If $\Delta \vdash_{amb} A \leq B$ then $\vdash \Delta$ and $\Delta \vdash A$ and $\Delta \vdash B$.

5.2.2 Subtyping

The subtyping relation is almost the same as the original rules by Amadio and Cardelli [5] in Figure 2.1. The noticeable difference is the addition of various well-formedness checks in various rules. For instance, base cases such as rule Amber-nat and rule Amber-top check whether the environments are well-formed. Moreover, in rule Amber-self we require the recursive type to be well-formed ($\Delta \vdash \mu\alpha.\, A$).

5.3 From the Amber Rules to the Specification

To prove the soundness theorem with respect to our new specification of iso-recursive subtyping, we need to prove that Amber subtyping is sound with respect to weakly positive subtyping and the double unfolding rules.

5.3.1 From the Amber rules to weakly positive subtyping

The first step is to translate the environments and types used in the Amber formulation, since they have different forms.

Definition 46. Translation of environments and types from the Amber rules.

$$|\cdot| = \cdot \qquad (\cdot)(A) = A$$
$$|\Delta,\ \alpha \leq \beta| = |\Delta|,\ \alpha \qquad (\Delta,\ \alpha \leq \beta)(A) = (\Delta)([\beta \mapsto \alpha]\ A)$$

The translation functions, $|\cdot|$ and $(\cdot)(A)$, simply drop every second variable defined in the context Δ. For example, a subtyping judgment in the Amber system $\alpha \leq \beta \vdash \alpha \to \top \leq \beta \to \top$ is translated to $\alpha \vdash \alpha \to \top \leq \alpha \to \top$.

Before showing the relationship between the Amber subtyping and our subtyping with the positive restriction, we must prove an important auxiliary lemma:

Lemma 47. If $\Delta \vdash_{amb} A \leq B$ and $(\alpha \leq \beta) \in \Delta$, then

1. $\alpha \notin fv(B)$ and $\beta \notin fv(A)$ implies $\alpha \in_+ (\Delta)(A) \leq (\Delta)(B)$ and
2. $\alpha \notin fv(A)$ and $\beta \notin fv(B)$ implies $\alpha \in_- (\Delta)(A) \leq (\Delta)(B)$.

Proof. Do induction on $\Delta \vdash_{amb} A \leq B$.

- Rule Amber-var: In such case $A = \alpha'$ and $B = \beta'$.
  - ⋆ Goal (1): If $\alpha \neq \alpha'$, we achieve the goal (recall that $\alpha \in_m \alpha' \leq \alpha'$ always holds for any mode m). Otherwise, we know that $\alpha = \alpha'$. Since $(\alpha \leq \beta) \in \Delta$, the goal becomes $\alpha \in_+ \alpha \leq \alpha$.
  - ⋆ Goal (2): $\alpha \notin fv(A)$ implies $\alpha \neq \alpha'$.
- Rule Amber-rec: Assume $A = \mu\alpha'.\ A'$ and $B = \mu\beta'.\ B'$, then the goal becomes $\alpha \in_m (\Delta)(\mu\alpha'.\ A') \leq (\Delta)(\mu\beta'.\ B')$ (m is $+$ and $-$, respectively, for two goals), which can be rewritten as $\alpha \in_m \mu\alpha'.\ (\Delta)(A') \leq \mu\beta'.\ (\Delta)(B')$.

  The induction hypotheses becomes:

  1. $(\alpha \leq \beta) \in \Delta, (\alpha' \leq \beta') \Rightarrow \alpha \notin fv(B') \Rightarrow \beta \notin fv(A') \Rightarrow \alpha \in_+ (\Delta)(A') \leq (\Delta)(B')$ and
  2. $(\alpha \leq \beta) \in \Delta, (\alpha' \leq \beta') \Rightarrow \alpha \notin fv(A') \Rightarrow \beta \notin fv(B') \Rightarrow \alpha \in_- (\Delta)(A') \leq (\Delta)(B')$.

  - ⋆ For goal (1): we apply rule Posvar-rec, then we need to check if $\alpha' \in_+ (\Delta)(A') \leq (\Delta)(B')$ and $\alpha \in_+ (\Delta)(A') \leq (\Delta)(B')$. Both cases can be solved by applying induction hypothesis (1).
  - ⋆ For goal (2): we apply rule Posvar-rec, then we need to check if $\alpha' \in_- (\Delta)(A') \leq (\Delta)(B')$ and $\alpha \in_+ (\Delta)(A') \leq (\Delta)(B')$. For the former one we apply induction hypothesis (2), and for the latter one we apply induction hypothesis (1).
- Rule Amber-self: Assume $A = B = \mu\alpha'.\ A'$. From the condition of the goal, we know that $\alpha \notin fv(A')$ always holds, thus $\alpha \in_m (\Delta)(A) \leq (\Delta)(B)$ is true for any mode m.

□

With the help of Lemma 47, we can prove that if two types are subtypes under the Amber rules, they are also subtypes under weakly positive subtyping:

Theorem 48. If $\Delta \vdash_{amb} A \leq B$ then $|\Delta| \vdash (\Delta)(A) \leq_+ (\Delta)(B)$.

Proof. Do induction on $\Delta \vdash_{amb} A \leq B$. We show only the more interesting case for recursive types.

- Rule Amber-rec: Assume $A = \mu\alpha'.\ A'$ and $B = \mu\beta'.\ B'$. The goal becomes $|\Delta| \vdash (\Delta)(\mu\alpha'.\ A') \leq (\Delta)(\mu\beta'.\ B')$, which can be rewritten as $|\Delta| \vdash \mu\alpha'.\ (\Delta)(A') \leq \mu\beta'.\ (\Delta)(B')$. Apply rule PosRes-rec, the first premise $|\Delta, \alpha \leq \beta| \vdash (\Delta)(A') \leq (\Delta)(B')$ can be solved by induction hypothesis. Since we know $\Delta, \alpha' \leq \beta' \vdash_{amb} A' \leq B'$, we apply Lemma 47 to it. By obtaining $\alpha' \in_+ (\Delta)(A') \leq (\Delta)(B')$, we can solve the second premise.

□

5.3.2 From weakly positive subtyping to double unfoldings

We are now one step away from the soundness theorem: to prove that the weakly positive subtyping implies double unfolding subtyping. The main difference is on rule PosRes-rec, which corresponds to rule SA-rec in the double unfolding subtyping. The proof requires the following lemma which reveals an important property to prove that the weakly positive subtyping implies double unfolding subtyping:

Lemma 49. If $\alpha \in_m A \leq B$ and $\beta \in_+ A \leq B$ then $\alpha \in_m [\beta \mapsto A]\ A \leq [\beta \mapsto B]\ B$.

This lemma tells us that the positive restriction respects the mode on non-negative substitutions.

The proof of Lemma 49, as other substitution lemmas we have showed before, requires a generalization. Such a generalization is a bit tricky, since we allow equal types in the positive restriction. For readers interested in the details of the generalization, we refer to our mechanized proof.

This lemma is important because it shows that, with Lemma 49 proved, we can derive the following lemma, which relates weakly positive subtyping to our algorithmic subtyping relation in the double unfolding form:

Lemma 50. If $\Gamma \vdash_a A \leq B$ and $\Gamma \vdash_a C \leq D$, then

1. $\alpha \in_+ A \leq B$ implies $\Gamma \vdash_a [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$ and
2. $\alpha \in_- B \leq A$ implies $\Gamma \vdash_a [\alpha \mapsto D]\ A \leq [\alpha \mapsto C]\ B$.

Proof. Do induction on $\Gamma \vdash_a A \leq B$. We only show how to prove the function case.

- Rule SA-arrow: Assume $A = A_1 \to A_2$ and $B = B_1 \to B_2$.
  - ⋆ Goal (1):

- The goal becomes $\Gamma \vdash_a [\alpha \mapsto C]\ (A_1 \to A_2) \leq [\alpha \mapsto D]\ (B_1 \to B_2)$, which can be rewritten as $\Gamma \vdash_a ([\alpha \mapsto C]\ A_1) \to ([\alpha \mapsto C]\ A_2) \leq ([\alpha \mapsto D]\ B_1) \to ([\alpha \mapsto D]\ B_2)$.
- The condition from Goal (1) becomes $\alpha \in_+ A_1 \to A_2 \leq B_1 \to B_2$. By inversion, we obtain $\alpha \in_- B_1 \leq A_1$ and $\alpha \in_+ A_2 \leq B_2$.
- Apply the constructor on the goal, we need to prove: $\Gamma \vdash_a [\alpha \mapsto D]\ B_1 \leq [\alpha \mapsto C]\ A_1$ and $\Gamma \vdash_a [\alpha \mapsto C]\ A_2 \leq [\alpha \mapsto D]\ B_2$.
- For the latter one, we apply the induction hypothesis (1).
- For the former one, we apply the induction hypothesis (2). However, we need to prove $\alpha \in_- A_1 \leq B_1$. Recall that the positive restriction is commutative, so from $\alpha \in_- B_1 \leq A_1$ we can prove $\alpha \in_- A_1 \leq B_1$.

⋆ Goal (2):

- The goal becomes $\Gamma \vdash_a [\alpha \mapsto D]\ (A_1 \to A_2) \leq [\alpha \mapsto C]\ (B_1 \to B_2)$, which can be rewritten as $\Gamma \vdash_a ([\alpha \mapsto D]\ A_1) \to ([\alpha \mapsto D]\ A_2) \leq ([\alpha \mapsto C]\ B_1) \to ([\alpha \mapsto C]\ B_2)$.
- The condition from Goal (1) becomes $\alpha \in_- B_1 \to B_2 \leq A_1 \to A_2$. By inversion, we obtain $\alpha \in_+ A_1 \leq B_1$ and $\alpha \in_- B_2 \leq A_2$.
- Apply the constructor on the goal, we need to prove: $\Gamma \vdash_a [\alpha \mapsto C]\ B_1 \leq [\alpha \mapsto D]\ A_1$ and $\Gamma \vdash_a [\alpha \mapsto D]\ A_2 \leq [\alpha \mapsto C]\ B_2$.
- For the latter one, we apply the induction hypothesis (2).
- For the former one, we apply the induction hypothesis (1). However, we need to prove $\alpha \in_+ B_1 \leq A_1$. Because the positive restriction is commutative, from $\alpha \in_+ A_1 \leq B_1$ we can prove $\alpha \in_+ B_1 \leq A_1$.

□

Corollary 51. If $\alpha \in_+ A \leq B$ and $\Gamma \vdash_a A \leq B$ then $\Gamma \vdash_a [\alpha \mapsto A]\ A \leq [\alpha \mapsto B]\ B$.

With the Corollary 51, the relation between positive restriction and the algorithmic double unfolding subtyping is easy to establish:

Theorem 52. If $\Gamma \vdash A \leq_+ B$ then $\Gamma \vdash_a A \leq B$.

5.3.3 The soundness theorem

Combining Lemmas 23, 48 and 52, we have

Corollary 53. Soundness of the Amber rules with respect to declarative formulation.

$$\text{If } \Delta \vdash_{amb} A \leq B \text{ then } |\Delta| \vdash (\Delta)(A) \leq (\Delta)(B).$$

5.4 From the Specification to the Amber Rules

The completeness theorem with respect to our new specification of iso-recursive subtyping, to some degree, is more difficult than soundness theorem. Because the Amber rules are more complex in terms of shape than the double unfolding rule, more cases need to be discussed when we do induction on a simpler formulation.

5.4.1 From double unfoldings to weakly positive subtyping

Firstly, let us consider how to convert the double unfolding rule to weakly positive subtyping. The double unfolding rule and weakly positive subtyping share the same context, which means the only source of difference comes from the treatment of recursive types. For weakly positive subtyping, the following inversion lemma is useful:

Lemma 54. If $\Gamma \vdash A \leq_+ B$ and $\Gamma \vdash C \leq_+ D$, then

1. $\Gamma \vdash [\alpha \mapsto C]\ A \leq_+ [\alpha \mapsto D]\ B$ implies $\alpha \in_+ A \leq B$ or $C = D$;
2. $\Gamma \vdash [\alpha \mapsto D]\ A \leq_+ [\alpha \mapsto C]\ B$ implies $\alpha \in_- A \leq B$ or $C = D$.

This lemma states that if after substitution the subtyping relation is preserved, then either C and D are equal; or the type variable respects the weakly positive restriction.

Now we can prove that weakly positive subtyping is complete with respect to the double unfolding formulation.

Theorem 55.

$$\text{If } \Gamma \vdash_a A \leq B \text{ then } \Gamma \vdash A \leq_+ B.$$

Proof. Induction on $\Gamma \vdash_a A \leq B$. All cases are straightforward except when A is $\mu\alpha.\ A'$ and B is $\mu\alpha.\ B'$. By induction hypothesis, we know that $\Gamma \vdash A' \leq_+ B'$. By applying lemma 54 with $A := A'$, $B := B'$, $C := A'$, $D := B'$, and mode $+$, we get that either $\alpha \in_+ A' \leq B'$ or $A' = B'$. For the former case, we apply constructor rule PosRes-rec. For the latter case, we apply reflexivity. □

5.4.2 Form weakly positive subtyping to the Amber rules

The translation from weakly positive subtyping to the Amber rules is quite tricky due to the different shapes of the contexts. To illustrate the difficulty consider the following subtyping statement using weakly positive subtyping:

$$\Gamma, \alpha \vdash \top \to \alpha \leq_+ \mathsf{nat} \to \alpha$$

where the environment binds the type variable α. For proving the subtyping relationship, we need to prove:

$$\Gamma, \alpha \vdash \alpha \leq_+ \alpha$$

$\boxed{\Gamma \vdash A \leq_+ B \rhd \Pi}$ *(Position Allocation)*

$$\frac{}{\cdot \vdash A \leq_+ B \rhd \cdot}\ \textsc{mono-nil} \qquad \frac{\alpha \in_m A \leq B \qquad \Gamma \vdash A \leq_+ B \rhd \Pi}{\Gamma, \alpha \vdash A \leq_+ B \rhd \Pi, m}\ \textsc{mono-cons}$$

Figure 5.3: Position allocation for weakly positive subtyping.

However, if we want to prove the same statement using the Amber rules, we need to change the relationship to:

$$\Delta, \alpha \leq \beta \vdash_{amb} \top \to \alpha \leq \mathsf{nat} \to \beta$$

and

$$\Delta, \alpha \leq \beta \vdash_{amb} \alpha \leq \beta$$

Note that, in weakly positive subtyping, we only need to store the free variables in the environment, while in the Amber rules, we have more variables and store the subtyping relationship between those variables as well.

The recipe of the conversion is to first generate a bundle of variables and match them to existing variables. Then we determine the mode for each variable in weakly positive subtyping, which helps us to allocate every pair of generated variables. After converting the context and types to the form of Amber rules, we prove that they preserve the subtyping relationship under the Amber rules.

As a second example, assume that we want to convert the following judgment into an Amber judgment

$$\alpha, \beta \vdash \beta \to \alpha \leq_+ \beta \to \alpha$$

we first generate new variables α', β' and assume the subtyping relations $\alpha \leq \alpha'$, $\beta \leq \beta'$. Then we examine the positivity of both variables and find out that these relations hold

$$\alpha \in_+ \beta \to \alpha \leq \beta \to \alpha \text{ and } \beta \in_- \beta \to \alpha \leq \beta \to \alpha$$

In the next step, we substitute the variables in the typing judgment, according to the mode and location. If the variable is in the left-hand side and occurs positively, or right-hand side and occurs negatively, we keep the variable as it is. Otherwise, we substitute the variable with its corresponding one ($\alpha \mapsto \alpha'$ and $\beta \mapsto \beta'$). After these steps, the final result becomes a normal Amber judgment, which has the same meaning of the initial judgment:

$$\alpha \leq \alpha', \beta \leq \beta' \vdash_{amb} \beta' \to \alpha \leq \beta \to \alpha'$$

We prove that this subtyping relation holds under the Amber rules.

Position Allocation As Figure 5.3 shows, we define a relation that relates each variable to a mode. The mode in Π has a one-to-one correspondence to the variables in Γ in the same order.

The definition of Π is

$$\Pi := \cdot \mid \Pi, + \mid \Pi, -$$

Note that it is not necessarily the case that Π is unique. For example, a variable that never occurs can be accepted by both modes, therefore its corresponding element in Π can be any mode.

Definition 56. Generation of a bundle of fresh variables.

$$\langle \Gamma \rangle := \{(\alpha \leq \beta) \mid \forall \alpha \in \Gamma,\ \beta \text{ is fresh}\}$$

After we have a list of pairs of variables (denoted as $\langle \Gamma \rangle$) and the mode for each variable, we design a function that converts the types according to our information. Note that $\langle \Gamma \rangle$ has same form as the contexts in the Amber setting.

Definition 57. We design a function $convert(\Delta, \Pi, A, m)$ for converting types from weakly positive subtyping setting to the Amber setting, which takes four inputs: a context for Amber formulation, a stack of modes, a type A and a mode. This function returns the converted type as output. Note that the Π is computed as Figure 5.3 shown, thus its length is equal to the length of Δ.

$$convert(\Delta, \Pi, A, m) = \begin{cases} A & \text{If } \Delta \text{ and } \Pi \text{ are empty.} \\ convert(\Delta', \Pi', [\alpha \mapsto \beta]\ A, m) & \text{If } \Delta = \Delta', \alpha \leq \beta \text{ and } \Pi = \Pi', m \\ convert(\Delta', \Pi', A, m) & \text{If } \Delta = \Delta', \alpha \leq \beta \text{ and } \Pi = \Pi', \overline{m} \end{cases}$$

We can now state the completeness theorem with Definition 57, where the subtyping relation of weakly positive subtyping preserves under the Amber rules.

Theorem 58. Completeness of the Amber rules: If $\Gamma \vdash A \leq_+ B$ and $\Gamma \vdash A \leq_+ B \rhd \Pi$, denoted $\langle \Gamma \rangle$ as Δ, then

$$\Delta \vdash_{amb} convert(\Delta, \Pi, A, -) \leq convert(\Delta, \Pi, B, +).$$

For simplicity, we skip the procedure of the proof for this theorem. Theorem 58 has a very long mechanized proof in the thesis, which relies on plenty of auxiliary lemmas distinguishing whether two recursive types are equal carefully.

The theorem involves some manipulation of the context and types, due to the inconsistency of contexts between our system and the Amber rules. However, it is very easy to obtain a simple form of corollary where the contexts are empty:

Corollary 59.

$$\text{If } \cdot \vdash A \leq_+ B \text{ then } \cdot \vdash_{amb} A \leq B.$$

The statement is less general than Theorem 58, but it does reveal that the programmer cannot distinguish between our algorithm and the Amber one, since in the subsumption rule,

the subtyping judgment always starts with an empty subtyping context. That is, type variables in the double unfolding formulations, and subtyping relations between type variables in the Amber formulations are only introduced by the subtyping relation, and not by the typing relation. The only information that should be in the context during the subsumption rule is the type information for variables.

5.4.3 The completeness theorem

Combining Lemma 19, Lemma 55 and Lemma 59, we have:

Corollary 60. Completeness of the Amber rules with respect to declarative formulation.

$$\text{If } \cdot \vdash A \leq B \text{ then } \cdot \vdash_{amb} A \leq B.$$

5.5 The Equivalence among All Formulations

So far, we have proven all directions of each equivalence. With the equivalence theorems, transitivity and unfolding lemma for our formulations (Lemmas 60, 53, 6 and 8), we can claim the Amber rules are transitive and satisfy the unfolding lemma.

Corollary 61. Transitivity of the Amber rules.

$$\text{If } \cdot \vdash_{amb} A \leq B \text{ and } \cdot \vdash_{amb} B \leq C \text{ then } \cdot \vdash_{amb} A \leq C.$$

Corollary 62. Unfolding lemma for the Amber rules.

$$\text{If } \cdot \vdash_{amb} \mu\alpha.\ A \leq \mu\alpha.\ B \text{ then } \cdot \vdash_{amb} [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B.$$

Notably, for transitivity, it is interesting to observe that transitivity holds under an empty environment. In Chapter 2.1.4, we discussed the issues with transitivity and showed a counter-example. That counter-example does not apply to our transitivity lemma because it uses non-empty environments. Therefore, a possible "fix" to the declarative formulation in Figure 5.2 is to restrict the transitivity rule to use only empty environments.

Finally, Table 5.1 and Figure 5.4 summarize some key lemmas and theorems until this chapter. In particular, it shows that all five formulations of subtyping presented in Chapter 3, 4 and 5 are equivalent in terms of expressive power.

5.6 Mechanized Proofs

The folder *coq_theory* includes all the Coq proofs about the weakly positive subtyping, a variant of Amber rules, and all equivalence shown in this chapter.

Table 5.1: The equivalence theorems in Part II.

	Reflexivity	Transitivity	Unfolding Lemma
Amber Rules	Built-in	Corollary 61	Corollary 62
Finite Unfolding	Theorem 5	Theorem 6	Lemma 8
Double Unfolding	Theorem 17	Theorem 18	Lemma 26
Nominal Unfolding	Theorem 27	Theorem 28	Lemma 29
Weakly Positive Subtyping	Theorem 41	Theorem 43	Lemma 44

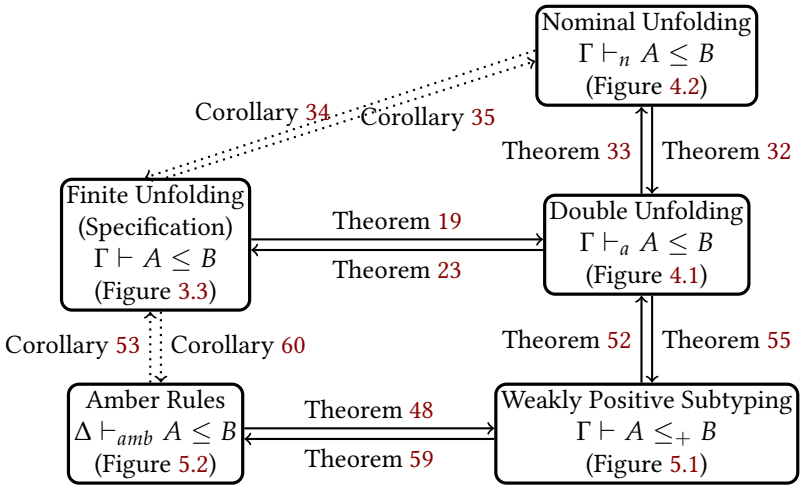


Figure 5.4: A diagram with the soundness and completeness lemmas in this work.

5.6.1 Definitions

All the definitions in the Chapter 5 can be found in file *AmberBase.v*. Table 5.2 shows the correspondence of definitions between the paper and the Coq artifacts. The file *AmberBase.v*, contains the definitions for the Amber rules and the intermediate subtyping relation based on a weakly positive restriction presented in this chapter.

5.6.2 Lemmas and theorems

Table 5.3 shows the descriptions for all the proof scripts in Chapter 5. For succinctness, we briefly describe the important lemmas and theorems.

5.6.3 Alternative weakly positive subtyping

During the proof of completeness of Amber rules, we found that the built-in reflexivity in the weakly positive subtyping disturbs the computation of position allocation for recursive types. Thus, in the mechanized proof, we use an alternative (Definition 63) for weakly positive subtyping to compute the mode more precisely: the default positive mode for equal recursive types.

The key idea is to unify rule PosRes-self and rule PosRes-rec into one rule, then we "hide" the problematic reflexivity subtly by rule PosRes-recalt:

Table 5.2: Paper-to-proofs correspondence guide in Chapter 5.

Definition	File	Name in Coq	Notation
Well-formed Type (Figure 5.2)	AmberBase.v	wf_amber E A	$\Delta \vdash A$
Amber Rules (Figure 5.2)	AmberBase.v	sub_amber E A B	$\Delta \vdash_{amb} A \leq B$
Weakly Positive Restriction (Figure 5.1)	AmberBase.v	posvar m X A B	$\alpha \in_m A \leq B$
Weakly Positive Subtyping (Figure 5.1)	PositiveBase.v	wk_sub E A B	$\Gamma \vdash A \leq_+ B$
Weakly Positive Subtyping (Definition 63)	AmberBase.v	sub_amber2 E A B	$\Gamma \vdash_u A \leq_+ B$

Table 5.3: Descriptions for the proof scripts in Chapter 5.

Theorems	Description	Files	Name in Coq
Theorem 41	Reflexivity (Positive)	AmberBase.v	sub_amber2_refl
Theorem 43	Transitivity (Positive)	PositiveBase.v	sub_amber2_trans
Lemma 44	Unfolding Lemma (Positive)	PositiveBase.v	unfolding_for_pos
Theorem 48	Amber to Positive	AmberBase.v	sub_amber_to_amber_2
Theorem 52	Positive to Double	AmberSoundness.v	sub_amber_2_to_sub
Corollary 53	Soundness (Amber)	AmberSoundness.v	amber_soundness2
Theorem 55	Double to Positive	PositiveSubtyping.v	sub_to_amber2
Theorem 59	Positive to Amber	AmberCompleteness.v	amber_complete_aux
Corollary 60	Completeness (Amber)	AmberCompleteness.v	amber_complete2
Corollary 61	Transitivity (Amber)	AmberCompleteness.v	amber_transitivity
Corollary 62	Unfolding Lemma (Amber)	AmberCompleteness.v	amber_unfolding

Definition 63. An alternative rule for checking if two recursive types are subtypes in weakly positive subtyping:

$$\textsc{PosRes-recalt}\quad \frac{\Gamma, \alpha \vdash A \leq_+ B \qquad \beta \text{ is fresh} \qquad \beta \in_+ \mu\alpha.\, A \leq \mu\alpha.\, B}{\Gamma \vdash \mu\alpha.\, A \leq_+ \mu\alpha.\, B}$$

Denoting $\Gamma \vdash_u A \leq_+ B$ as the weakly positive subtyping with the alternative rule PosRes-recalt for recursive types, we show that it has same expressiveness as the original definition of weakly positive subtyping (Figure 5.1):

Lemma 64. The two representations of weakly positive subtyping are equivalent:

$$\Gamma \vdash_u A \leq_+ B \Leftrightarrow \Gamma \vdash A \leq_+ B.$$

Proof. We actually need to prove both directions:

- ($\Rightarrow$): Induction on $\Gamma \vdash_u A \leq_+ B$.
  - Rule PosRes-recalt: Assume $A = \mu\alpha.\, A'$ and $B = \mu\alpha.\, B$.

* We know $\Gamma, \alpha \vdash_u A' \leq_+ B'$, and from hypothesis induction, we can derive $\Gamma, \alpha \vdash A' \leq_+ B'$.
* Among the conditions we also know $\beta \in_+ \mu\alpha.\ A' \leq \mu\alpha.\ B'$. Do inversion on it, we get two sub-cases:
  · Rule PosVar-rec: we have the condition $\alpha \in_+ A' \leq B'$, thus we apply rule PosRec-rec.
  · Rule PosVar-recself: the goal becomes $\Gamma \vdash \mu\alpha.\ A' \leq_+ \mu\alpha.\ B'$. Apply rule PosRes-self.

- ($\Leftarrow$): Induction on $\Gamma \vdash A \leq_+ B$.
  – Rule PosRes-rec: Assume $A = \mu\alpha.\ A'$ and $B = \mu\alpha.\ B'$.
    * We know $\Gamma, \alpha \vdash A' \leq_+ B'$, and from hypothesis induction, we can derive $\Gamma, \alpha \vdash_u A' \leq_+ B'$.
    * We also know $\alpha \in_+ A' \leq B'$. Our goal now is $\beta \in_+ \mu\alpha.\ A' \leq \mu\alpha.\ B'$.
    * Apply constructor rule PosVar-recalt, then the first premise $\beta \in_+ A' \leq B'$ is always held because β is fresh. The second premise is $\alpha \in_+ A' \leq B'$.
  – Rule PosRes-self: Apply Theorem 41.

□

5.6.4 Variable generation

Another difficulty worth mentioning is generating a bundle of variables in Definition 56. Such definition actually does two things: (1) generate a set of fresh variables; (2) match every fresh variable with an existing variable. This is a bit involved in Coq.

File *AmberCompleteness.v* gives the details showing how to solve this issue. We iterate each variable (denote as α) in context Γ, generate a fresh variable β and store both variables. One possibility is that the name of α might be used in previous stored set of variables. In that case, we generate one more fresh variable and store it. After that, we have a set of mixed variables containing all variables in context Γ and the number of new fresh variables is the same as the size of context Γ. All the variables in the set are distinct. Then we filter variables that belong to Γ and match them with variables in Γ one by one. Finally, we have a valid $\langle\Gamma\rangle$, as Definition 56 describes.

Part III

Extensions

Chapter 6

Non-Antisymmetric Subtyping

So far we considered calculi where the subtyping relation is antisymmetric. For instance, for the calculus presented in Chapter 3 and 4, Lemmas 21 and 24 hold. Both the Amber rules and the new rules proposed by us work well for antisymmetric subtyping relations. However, as explained in Chapter 2.1.4, applying the Amber rules in subtyping relations that are not antisymmetric is non-trivial due to the built-in reflexivity rule. The purpose of this section is to show that, unlike the Amber rules, the double/nominal unfolding rules can be easily applied to subtyping relations that are not antisymmetric. In this chapter we show the type soundness for two extensions of the calculus in Chapter 4. One is with records and records types, and the other one is with intersection types. Both type systems lead to a subtyping relation that is not antisymmetric. An important finding in this chapter is that, for the system with intersection types, the double unfolding and nominal unfolding rules are *not equivalent*. For subtyping with intersection types the nominal unfolding rules work well, but the problem of spurious subtyping in the double unfolding rules (see Chapter 6.3) cannot be prevented with the extra 1-time finite unfolding premise.

6.1 Overview

In Chapter 2.1.4, we have discussed that the Amber rules cannot deal well with some forms of subtyping. In particular, the reflexivity rule is limiting when the subtyping relation is not antisymmetric. In the context of subtyping, antisymmetry is the property that if two types are both subtypes of each other, then the two types are (syntactically) equal. More formally:

$$\Gamma \vdash A \leq B \wedge \Gamma \vdash B \leq A \Rightarrow A = B$$

In simple subtyping relations, such as for instance a simply typed lambda calculus extended with the top type and recursive types, this property holds. For instance, the calculus in Chapter 3 has an antisymmetric subtyping relation.

Unfortunately, many languages contain subtyping relations that are not antisymmetric. For instance, if a language contains some form of record types (which includes essentially all OOP languages), then the subtyping relation is not antisymmetric. In the example below, the

subtyping statement

$$\mu\alpha.\ \{x : \alpha, y : \mathsf{nat}\} \to \mathsf{nat} \leq \mu\alpha.\ \{y : \mathsf{nat}, x : \alpha\} \to \mathsf{nat}$$

should hold, since $\{x : \alpha, y : \mathsf{nat}\}$ and $\{y : \mathsf{nat}, x : \alpha\}$ are subtypes of each other. However, the two types are not syntactically equal. In such a setting, the use of the Amber rules would require that, instead of using syntactic equality in the reflexivity rule, we should use an equivalence relation on types. However, we cannot simply define equivalence to be:

$$\Gamma \vdash A \sim B \ := \ \Gamma \vdash A \leq B \wedge \Gamma \vdash B \leq A$$

because then the reflexivity rule would become (by a simple unfolding of the equivalence definition):

$$\frac{\Delta \vdash A \leq B \quad \Delta \vdash B \leq A}{\Delta \vdash A \leq B}\text{Amber-Refl-Wrong}$$

which would lead to a circular (and ill-behaved) subtyping relation. Instead, a separate equivalence relation needs to be defined to ensure that record types are equivalent up-to permutation. But adding such a separate relation on types would add complexity, since we would need a new set of rules and theorems about such relation.

In contrast, with the formulations based on the double unfoldings, because reflexivity is not built-in, we can simply define the equivalence relation above ($\Gamma \vdash A \sim B$) via subtyping. Thus, the double and nominal unfolding rules do not require a separate definition of equivalence, and they also do not rely on the subtyping relation being antisymmetric.

The calculus in this chapter illustrates the addition of records and records types to the calculus in Chapter 4. This addition has minimal impact of the calculus and metatheory: the proof techniques are similar, except that instead of syntactic equality we use our equivalence definition for types when proving the unfolding lemma.

When modelling the abstract syntax of types, we can pick a data structure, such as a finite map, that gives us the properties that we want for record types (and records) for free. That is, we can define record types to be something like:

$$\{\text{String} \mapsto \text{Type}\}$$

instead of a list of pairs of Strings and Types. Using lists introduces accidental complexity because order becomes relevant, and we need to ensure that the field names are distinct. For record types, a better choice of representation could help. However, presentations of records using sequences are quite common. Our own presentation in Chapter 6.2 is based on Pierce's one in *Types and Programming Language* [100], and he basically uses a list of pairs for his presentation and implementations.

Regarding more interesting examples of antisymmetry, we believe that type systems with binary intersection types would be an example. In those type systems we wish to have $A \wedge B$

and $B \wedge A$ to be equivalent types, for example. Unlike the case for records, a clever representation of syntax does not seem to help in such systems. Therefore, besides records and records types, we also illustrate the addition of intersection types to the calculus in Chapter 4.

6.2 The Formalization of Recursive Record Types

As explained in Chapter 6.1, applying the Amber rules in subtyping relations that are not antisymmetric is non-trivial due to the built-in reflexivity rule. The purpose of this section is to show that, unlike the Amber rules, the double unfolding rules can be easily applied to subtyping relations that are not antisymmetric. In this section we show the type soundness for an extension of the calculus in Chapter 4 with records and records types, which leads to a subtyping relation that is not antisymmetric when record types are represented as a sequence of pairs of labels and types.

6.2.1 Syntax and well-formedness

The syntax of the calculus is:

Types	A, B, C, D	$::=$	$\mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\, A \mid \boldsymbol{\{l_i : A_i}^{\,i \in 1 \cdots n}\boldsymbol{\}}$
Expressions	e	$::=$	$x \mid \mathsf{i} \mid e_1\, e_2 \mid \lambda x : A.\, e \mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e \mid$ $\boldsymbol{\{l_i = e_i}^{\,i \in 1 \cdots n}\boldsymbol{\}} \mid \boldsymbol{e.l}$
Values	v	$::=$	$\mathsf{i} \mid \lambda x : A.\, e \mid \mathsf{fold}\ [A]\ v \mid \boldsymbol{\{l_i = v_i}^{\,i \in 1 \cdots n}\boldsymbol{\}}$

Natural numbers, arrow types, the top type, type variables and recursive types are the same as before (Chapter 3.2). The additional syntax related to records and record types is highlighted with a bold font. The notation of record types is $\{l_i : A_i\ ^{i \in 1 \cdots n}\}$. Every label represents a type and all labels are required to be distinct. A record expression has the form of $\{l_i = e_i\ ^{i \in 1 \cdots n}\}$, and $e.l$ is the record projection expression.

In the type system with record types, we use $\Gamma \vdash A$ to represent that A is well-formed. The rules of $\Gamma \vdash A$ include most of the rules at the top of Figure 3.3. The rule WFT-RCD is new and ensures the well-formedness of record types. Similarly to Chapter 4, we use rule WFT-RECUR for recursive types.

6.2.2 Subtyping

Our subtyping rules follow the rules in Figure 4.1, but are extended with an algorithmic formulation of record subtyping. The definition of record subtyping (rule SA-RCD) is standard [100]: a record type A is a subtype of another record type B when: 1) all the labels in A are a subset of the labels in B; and 2) the field types of the corresponding labels are subtypes.

After adding record types, reflexivity and transitivity are still preserved.

Theorem 65. Reflexivity

$$\text{If } \vdash \Gamma \text{ and } \Gamma \vdash A \text{ then } \Gamma \vdash_a A \leq A.$$

$\boxed{\Gamma \vdash A}$ *(Well-Formed Type (with Record Types))*

$$\text{WFT-RECUR}\ \frac{\Gamma, \alpha \vdash A \qquad \Gamma, \alpha \vdash [\alpha \mapsto A]\ A}{\Gamma \vdash \mu\alpha.\ A} \qquad \text{WFT-RCD}\ \frac{\Gamma \vdash A_i \qquad \text{for each i}}{\Gamma \vdash \{l_i : A_i\ ^{i \in 1 \cdots n}\}}$$

$\boxed{\Gamma \vdash_a A \leq B}$ *(Subtyping)*

$$\text{SA-NAT}\ \frac{\vdash \Gamma}{\Gamma \vdash_a \mathsf{nat} \leq \mathsf{nat}} \qquad \text{SA-TOP}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash_a A \leq \top} \qquad \text{SA-VAR}\ \frac{\vdash \Gamma \qquad \alpha \in \Gamma}{\Gamma \vdash_a \alpha \leq \alpha}$$

$$\text{SA-ARROW}\ \frac{\Gamma \vdash_a B_1 \leq A_1 \qquad \Gamma \vdash_a A_2 \leq B_2}{\Gamma \vdash_a A_1 \to A_2 \leq B_1 \to B_2} \qquad \text{SA-REC}\ \frac{\Gamma, \alpha \vdash_a A \leq B \qquad \Gamma, \alpha \vdash_a [\alpha \mapsto A]\ A \leq [\alpha \mapsto B]\ B}{\Gamma \vdash_a \mu\alpha.\ A \leq \mu\alpha.\ B}$$

$$\text{SA-RCD}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \{k_j : A_j\ ^{j \in 1 \cdots m}\} \qquad \{l_i\ ^{i \in 1 \cdots n}\} \subseteq \{k_j\ ^{j \in 1 \cdots m}\} \qquad k_j = l_i \text{ implies } \Gamma \vdash_\alpha A_j \leq B_i}{\Gamma \vdash_a \{k_j : A_j\ ^{j \in 1 \cdots m}\} \leq \{l_i : B_i\ ^{i \in 1 \cdots n}\}}$$

Figure 6.1: Well-formedness and subtyping rules for record types.

Theorem 66. Transitivity

$$\text{If } \Gamma \vdash_a A \leq B \text{ and } \Gamma \vdash_a B \leq C \text{ then } \Gamma \vdash_a A \leq C.$$

Unlike the proof for the unfolding lemma in Chapter 4, we cannot rely on the antisymmetry lemma (Lemma 24) for proving the unfolding lemma. Instead of alpha-equivalence or syntactic equality, we introduce a weaker form of equivalence.

Definition 67 (Equivalence)**.**

$$\Gamma \vdash_a A \sim B \ := \ \Gamma \vdash_a A \leq B \wedge \Gamma \vdash_a B \leq A$$

With Definition 67, two record types $\{x : Int, y : Bool\}$ and $\{y : Bool, x : Int\}$ are considered to be equivalent: the only difference between two types is that one type is a permutation of the other type. In other words, the equivalence shows that the order in which the labels appear in a record type does not matter. One essential lemma is

Lemma 68. If $\Gamma \vdash_a A \leq B$ and $\Gamma \vdash_a C \sim D$, then $\Gamma \vdash_a [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$.

This lemma states that if two types are subtypes, then after substituting a recursive type variable α with two equivalent types, the subtyping relationship is preserved. The proof of this lemma is straightforward. With Lemma 68, we can prove our core lemma, as we did before.

Lemma 69. If

1. $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$;
2. $\Gamma_1, \alpha, \Gamma_2 \vdash_a C \leq D$;

3. $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\ C \leq \mu\alpha.\ D$;

then

1. $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$ and
2. $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto D]\ A \leq [\alpha \mapsto C]\ B$ implies $\Gamma_1, \Gamma_2 \vdash_a [\alpha \mapsto \mu\alpha.\ D]\ A \leq [\alpha \mapsto \mu\alpha.\ C]\ B$.

Proof. By induction on $\Gamma_1, \alpha, \Gamma_2 \vdash_a A \leq B$. Other cases are the same as proof of Lemma 25, except for:

- Case rule SA-VAR: In such case $A = B = \beta$. If $\alpha \neq \beta$, the goal is trivial.
  - Otherwise, for goal (1), we want to prove $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\ C \leq \mu\alpha.\ D$, which is actually premise (3).
  - For goal (2), we have $\Gamma_1, \alpha, \Gamma_2 \vdash_a C \leq D$ from premise (2), and $\Gamma_1, \alpha, \Gamma_2 \vdash_a D \leq C$ from the condition of goal (2), thus $\Gamma_1, \alpha, \Gamma_2 \vdash_a C \sim D$. By Lemma 68, we get $\Gamma_1, \alpha, \Gamma_2 \vdash_a [\alpha \mapsto D]\ D \leq [\alpha \mapsto C]\ C$. As a result, we have $\Gamma_1, \Gamma_2 \vdash_a \mu\alpha.\ D \leq \mu\alpha.\ C$.

□

Finally, we can prove the unfolding lemma:

Lemma 70. Unfolding Lemma

$$\text{If } \Gamma \vdash_a \mu\alpha.\ A \leq \mu\alpha.\ B \text{ then } \Gamma \vdash_a [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B.$$

A final remark is that the same technique that we employ here to prove the unfolding lemma could have been used in the calculus in Chapter 4 as well. In other words, we do not need to rely on the antisymmetry lemmas in Chapter 4. We opted to present the two techniques in the paper to also emphasize the difference between antisymmetric and non-antisymmetric relations, since for the Amber rules such difference is quite important.

6.2.3 Type safety

We use the same typing and reduction rules as Chapter 3, extended with extra rules for records and record types.

As the top of Figure 6.2 shows, we have two typing rules for record types. Rule TYPING-RCD states that a record is well-typed if we know that all its fields are well-typed. Rule TYPING-PROJ checks that the record that we are projecting from is well-typed, and contains the field label that we are projecting.

As the bottom of Figure 6.2 shows, we have three reduction rules for record types. Rule STEP-PROJRCD retrieves a component of a record. Rule STEP-PROJ reduces the record expression being projected. Rule STEP-RCD implements a left-to-right evaluation order to reduce a record.

$\boxed{\Gamma \vdash e : A}$ *(Typing)*

$$\textsc{typing-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{i} : \mathsf{nat}} \qquad \textsc{typing-var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{typing-sub}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}$$

$$\textsc{typing-abs}\ \frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash \lambda x : A_1.\ e : A_1 \to A_2} \qquad \textsc{typing-app}\ \frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2}$$

$$\textsc{typing-unfold}\ \frac{\Gamma \vdash e : \mu\alpha.\ A}{\Gamma \vdash \mathsf{unfold}\ [\mu\alpha.\ A]\ e : [\alpha \mapsto \mu\alpha.\ A]\ A} \qquad \textsc{typing-fold}\ \frac{\Gamma \vdash e : [\alpha \mapsto \mu\alpha.\ A]\ A \qquad \Gamma \vdash \mu\alpha.\ A}{\Gamma \vdash \mathsf{fold}\ [\mu\alpha.\ A]\ e : \mu\alpha.\ A}$$

$$\textsc{typing-rcd}\ \frac{\text{for each } i \quad \Gamma \vdash e_i : A_i}{\Gamma \vdash \{l_i = e_i\ ^{i \in 1 \cdots n}\} : \{l_i : A_i\ ^{i \in 1 \cdots n}\}} \qquad \textsc{typing-proj}\ \frac{\Gamma \vdash e : \{l_i : A_i\ ^{i \in 1 \cdots n}\}}{\Gamma \vdash e.l_i : A_i}$$

$\boxed{e_1 \hookrightarrow e_2}$ *(Reduction)*

$$\textsc{step-beta}\ \frac{}{(\lambda x : A.\ e_1)\ v_2 \hookrightarrow [x \mapsto v_2]\ e_1} \qquad \textsc{step-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2} \qquad \textsc{step-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\ e_2 \hookrightarrow v_1\ e_2'}$$

$$\textsc{step-fld}\ \frac{}{\mathsf{unfold}\ [A]\ (\mathsf{fold}\ [B]\ v) \hookrightarrow v} \qquad \textsc{step-unfold}\ \frac{e \hookrightarrow e'}{\mathsf{unfold}\ [A]\ e \hookrightarrow \mathsf{unfold}\ [A]\ e'} \qquad \textsc{step-fold}\ \frac{e \hookrightarrow e'}{\mathsf{fold}\ [A]\ e \hookrightarrow \mathsf{fold}\ [A]\ e'}$$

$$\textsc{step-projrcd}\ \frac{}{\{l_i = v_i\ ^{i \in 1 \cdots n}\}.l_j \hookrightarrow v_j} \qquad \textsc{step-proj}\ \frac{e \hookrightarrow e'}{e.l_j \hookrightarrow e'.l_j}$$

$$\textsc{step-rcd}\ \frac{e_j \hookrightarrow e_j'}{\{l_i = v_i\ ^{i \in 1 \cdots j-1},\ l_j = e_j,\ l_k = e_k\ ^{k \in j+1 \cdots n}\} \hookrightarrow \{l_i = v_i\ ^{i \in 1 \cdots j-1},\ l_j = e_j',\ l_k = e_k\ ^{k \in j+1 \cdots n}\}}$$

Figure 6.2: Typing and reduction rules for record types

The proof technique of proving type soundness is conventional, without any special approach, except for the use of the unfolding lemma in preservation (just as in Chapter 4). Therefore, we can directly prove preservation and progress.

Theorem 71. Preservation.

$$\text{If } \Gamma \vdash e : A \text{ and } e \hookrightarrow e' \text{ then } \Gamma \vdash e' : A.$$

Theorem 72. Progress.

$$\text{If } \vdash e : A \text{ then } e \text{ is a value or exists } e', e \hookrightarrow e'.$$

6.3 The Spurious Subtyping Problem, Revisited

In Chapter 4.1, we have briefly mentioned that compared to the double unfolding rule, the nominal unfolding avoids the spurious subtyping problem.

In some simple calculi like STLC, such spurious subtyping problem will not cause failure of the type soundness. Unfortunately, the simple approach employed by rule S-DOUBLE to avoid spurious subtyping does not work for calculi with intersection types. We illustrate the issue via a counter-example showing that the unfolding lemma does not hold with intersection types. Because the unfolding lemma does not hold, type preservation does not hold either.

6.3.1 Intersection subtyping

Intersection types are used to express that a value has multiple types. Subtyping relations with intersection types only have been unproblematic and their metatheory is well-established [16, 14, 99]. It enables the languages to employ *finite polymorphism* [99].

Before showing the counter-example we recall the basic subtyping rules for intersection types [16]:

$$\text{S-AND}\quad \frac{\Gamma \vdash A \leq B_1 \qquad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \& B_2} \qquad \text{S-ANDL}\quad \frac{\Gamma \vdash A_2 \qquad \Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \& A_2 \leq B} \qquad \text{S-ANDR}\quad \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \& A_2 \leq B}$$

Rule S-AND states that a type is a subtype of an intersection type $B_1 \& B_2$, if it is the subtype of all components (B_1 and B_2). Rules S-ANDL and S-ANDR state that an intersection type is the subtype of a type B, as long as at least one of the components is a subtype of B. An important point is that, with intersection subtyping, subtyping is no longer syntax-directed. For example, both rule S-ANDL and rule S-ANDR can be used to derive the subtyping statement $A \& A \leq A$.

6.3.2 The counter-example

Due to the fact the subtyping derivations with intersection types are not unique, we can construct a counter-example for the unfolding lemma. One of the counter-examples is when we have $A := \mu\alpha.\ \mathsf{nat} \& (\mathsf{nat} \to \mathsf{nat}) \& (\alpha \to \mathsf{nat})$ and $B := \mu\alpha.\ \mathsf{nat} \& (\alpha \to \mathsf{nat})$[1]. For convenience, here we denote A_1 and B_1 as, respectively, the 1-time finite unfolding of A and B; and A_2 and B_2 as, respectively, the 2-times finite unfolding of A and B. That is:

$$\begin{array}{lcl} A_1 & := & \mathsf{nat} \& (\mathsf{nat} \to \mathsf{nat}) \& (\alpha \to \mathsf{nat}) \\ B_1 & := & \mathsf{nat} \& (\alpha \to \mathsf{nat}) \\ A_2 & := & [\alpha \mapsto A_1]\ A_1 = \mathsf{nat} \& (\mathsf{nat} \to \mathsf{nat}) \& (A_1 \to \mathsf{nat}) \\ B_2 & := & [\alpha \mapsto B_1]\ B_1 = \mathsf{nat} \& (B_1 \to \mathsf{nat}) \end{array}$$

Type A should not be the subtype of type B, because the unfoldings of A and B are not subtypes. In other words, if $A \leq B$ then the unfolding lemma does not hold. The following derivation shows that *the unfoldings of A and B are not subtypes*:

[1] Note that we assume that the intersection operator & has the highest priority, both compared to the recursive type binder and function types. In particular, $\mu\alpha.\ A \& B$ is equivalent to $\mu\alpha.\ (A \& B)$.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{nat} \leq \mathsf{nat}}}{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(A \to \mathsf{nat}) \leq \mathsf{nat}}\ \textsc{s-andl} \quad \dfrac{\dfrac{\dfrac{B \leq A \text{ (fail!)} \quad \mathsf{nat} \leq \mathsf{nat}}{A \to \mathsf{nat} \leq B \to \mathsf{nat}}\ \textsc{s-arrow}}{(\mathsf{nat} \to \mathsf{nat})\&(A \to \mathsf{nat}) \leq B \to \mathsf{nat}}\ \textsc{s-andr}}{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(A \to \mathsf{nat}) \leq B \to \mathsf{nat}}\ \textsc{s-andr}}{\dfrac{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(A \to \mathsf{nat}) \leq \mathsf{nat}\&(B \to \mathsf{nat})}{[\alpha \mapsto A]\ A_1 \leq [\alpha \mapsto B]\ B_1}}\ \textsc{s-and}$$

In the derivation above (and later), we highlight important portions. We use colors for three types, two on an intersection on the left (the color red and blue, respectively), and one type on the right. Note that if the type on the right has a different color from a type on the left then the two types are not related by subtyping. In the derivation above the important point to notice is that $\mathsf{nat} \to \mathsf{nat} \not\leq B \to \mathsf{nat}$, since a recursive type is not subtype of a natural number ($B \not\leq \mathsf{nat}$). That is, attempting to use rule S-ANDL (instead of rule S-ANDR) would soon fail. We will come back to this point soon.

Unfortunately, with the double unfolding rules, $A \leq B$ does hold, breaking the unfolding lemma. For showing this fact, according to the rule S-DOUBLE, we just need to check both the 1-time and 2-times finite unfoldings for $A \leq B$.

It is easy to show that *the 1-time finite unfolding ($A_1 \leq B_1$) holds*:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{nat} \leq \mathsf{nat}}}{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(\alpha \to \mathsf{nat}) \leq \mathsf{nat}}\ \textsc{s-andl} \quad \dfrac{\dfrac{\alpha \to \mathsf{nat} \leq \alpha \to \mathsf{nat}}{(\mathsf{nat} \to \mathsf{nat})\ \&(\alpha \to \mathsf{nat}) \leq \alpha \to \mathsf{nat}}\ \textsc{s-andr}}{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(\alpha \to \mathsf{nat}) \leq \alpha \to \mathsf{nat}}\ \textsc{s-andr}}{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(\alpha \to \mathsf{nat}) \leq \mathsf{nat}\&(\alpha \to \mathsf{nat})}\ \textsc{s-and}$$

Here we note that the 1-time derivation follows a similar structure to that of the (failed) derivation for the unfoldings of A and B. In particular, we make similar choices with respect rules S-ANDR and S-ANDL.

Furthermore, *the 2-times finite unfolding ($A_2 \leq B_2$) also holds*:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{nat} \leq \mathsf{nat}}}{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(A_1 \to \mathsf{nat}) \leq \mathsf{nat}}\ \textsc{s-andl} \quad \dfrac{\dfrac{\dfrac{B_1 \leq \mathsf{nat} \quad \mathsf{nat} \leq \mathsf{nat}}{\mathsf{nat} \to \mathsf{nat} \leq B_1 \to \mathsf{nat}}\ \textsc{s-arrow}}{(\mathsf{nat} \to \mathsf{nat})\ \&(\ A_1 \to \mathsf{nat}\ ) \leq B_1 \to \mathsf{nat}}\ \textsc{s-andl}}{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(A_1 \to \mathsf{nat}) \leq B_1 \to \mathsf{nat}}\ \textsc{s-andr}}{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(A_1 \to \mathsf{nat}) \leq \mathsf{nat}\&(B_1 \to \mathsf{nat})}\ \textsc{s-and}$$

Let us review why this counter-example passes the subtyping check. The subtyping between the unfoldings of A and B gets stuck when we attempt to check $A \to \mathsf{nat} \leq B \to \mathsf{nat}$. Since both A and B are in a contravariant position, we need to flip the subtyping relation, and $B \leq A$ does not hold. As a result, $A \to \mathsf{nat} \not\leq B \to \mathsf{nat}$ and the derivation fails.

During the double finite unfolding, we expect that, similarly to the unfolding of the recursive types A and B, the function cases lead to a failed derivation, since $A_1 \to \mathsf{nat} \not\leq B_1 \to \mathsf{nat}$. However, those relations are derived from intersection types, which allow us to choose another

derivation. While in the unfolding of the recursive types A and B, those alternative derivations immediately fail, in the double finite unfolding they succeed as $\mathsf{nat} \to \mathsf{nat} \leq B_1 \to \mathsf{nat}$ holds (recall that $B_1 := \mathsf{nat}\&(\alpha \to \mathsf{nat})$). The key point is that such alternative derivation (via rule S-ANDL) should not hold. For instance in $\mathsf{nat} \to \mathsf{nat} \leq B_1 \to \mathsf{nat}$, the origin of the type B_1 is the result of a substitution of a recursive type variable by B_1. The information that is lost in the double unfolding is that the origin of type B_1 is from an unfolding substitution, and such type should not be comparable to regular types. In the derivation of unfolding lemma for the recursive types A and B, such information is present and what we have instead is $\mathsf{nat} \to \mathsf{nat} \not\leq B \to \mathsf{nat}$, which fails.

In summary, a form of spurious subtyping leads to a valid subtyping derivation with the double unfolding rules. Unlike the setting without intersection types, the 1-time unfolding is unable to prevent such form of spurious subtyping. The key difference is that the presence of intersection types offers a choice in how to proceed with the derivation, greatly complicating the detection of spurious subtyping. This fatal issue breaks the preservation theorem, which relies on the unfolding lemma. In Chapter 4.1.2 we already show a counter-example to the unfolding lemma that we have presented is also a counter-example of type preservation.

6.4 Intersection Subtyping with Nominal Unfoldings

With the nominal unfolding rule, the spurious subtyping problem can be solved nicely. For the counter-example above, we will show that

$$\mu\alpha.\ \mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(\alpha \to \mathsf{nat}) \not\leq \mu\alpha.\ \mathsf{nat}\&(\alpha \to \mathsf{nat})$$

works well with the nominal unfolding rules.

The failed subtyping derivation for the counter-example using the nominal unfolding rules is:

$$\dfrac{\dfrac{\dfrac{\mathsf{nat} \leq \mathsf{nat}}{\mathsf{nat}\&\ldots\&\ldots \leq \mathsf{nat}}\ \text{S-ANDL} \qquad \dfrac{\dfrac{\dfrac{B_1 \leq A_1\ \text{(fails!)}}{B_1^\alpha \leq A_1^\alpha} \qquad \mathsf{nat} \leq \mathsf{nat}}{A_1^\alpha \to \mathsf{nat} \leq B_1^\alpha \to \mathsf{nat}}\ \text{S-ARROW}}{(\mathsf{nat} \to \mathsf{nat})\ \&\ (A_1^\alpha \to \mathsf{nat}) \leq B_1^\alpha \to \mathsf{nat}}\ \text{S-ANDR}}{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(A_1^\alpha \to \mathsf{nat}) \leq B_1^\alpha \to \mathsf{nat}}\ \text{S-ANDR}}{\mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(A_1^\alpha \to \mathsf{nat}) \leq \mathsf{nat}\&(B_1^\alpha \to \mathsf{nat})}\ \text{S-AND}}{\mu\alpha.\ \mathsf{nat}\&(\mathsf{nat} \to \mathsf{nat})\&(\alpha \to \mathsf{nat}) \leq \mu\alpha.\ \mathsf{nat}\&(\alpha \to \mathsf{nat})}\ \text{S-REC}$$

Now the derivation tree is what we expect. After adding a label, the function type $\mathsf{nat} \to \mathsf{nat}$ does not match with the function type $B_1^\alpha \to \mathsf{nat}$, because the labelled type B_1^α is not a subtype of nat. Thus, we avoid an accidental subtyping derivation using rule S-ANDL.

$\boxed{\Gamma \vdash A}$ *(Well-Formed Type)*

$$\textsc{wft-nat}\ \frac{}{\Gamma \vdash \mathsf{nat}} \qquad \textsc{wft-Top}\ \frac{}{\Gamma \vdash \top} \qquad \textsc{wft-Bot}\ \frac{}{\Gamma \vdash \bot} \qquad \textsc{wft-var}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{wft-arrow}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2} \qquad \textsc{wft-flabel}\ \frac{\Gamma \vdash A}{\Gamma \vdash A^{\alpha}}$$

$$\textsc{wft-rec}\ \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \mu\alpha.\ A} \qquad \textsc{wft-and}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}$$

$\boxed{\Gamma \vdash A \leq B}$ *(Subtyping)*

$$\textsc{S-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nat} \leq \mathsf{nat}} \qquad \textsc{S-top}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash A \leq \top} \qquad \textsc{S-bot}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash \bot \leq A} \qquad \textsc{S-flabel}\ \frac{\Gamma \vdash A \leq B}{\Gamma \vdash A^{\alpha} \leq B^{\alpha}}$$

$$\textsc{S-arrow}\ \frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2} \qquad \textsc{S-fnominal}\ \frac{\Gamma, \alpha \vdash [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B} \qquad \textsc{S-var}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \alpha}{\Gamma \vdash \alpha \leq \alpha}$$

$$\textsc{S-and}\ \frac{\Gamma \vdash A \leq B_1 \qquad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \& B_2} \qquad \textsc{S-andL}\ \frac{\Gamma \vdash A_2 \qquad \Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \& A_2 \leq B} \qquad \textsc{S-andR}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \& A_2 \leq B}$$

Figure 6.3: Well-formedness and subtyping rules.

6.5 The Calculus with Intersection Types

In this section we present a subtyping relation with iso-recursive subtyping and intersection types. Unlike the problematic approach with the double unfolding rules (as discussed in Chapter 6.3), our new approach with nominal unfoldings works well and retains desirable properties. Among others, we prove *transitivity* of subtyping, the *unfolding lemma* and *decidability* of subtyping.

6.5.1 Syntax and well-formedness

The syntax of types, expressions, values and contexts is shown below, and the definition of well-formed types is shown at the top of Figure 6.3. Compared with the definition of types in Chapter 3.2, we just add intersection types and label types. Compared with the definition of well-formedness in Figure 4.2, we just add intersection types.

Types	A, B	$::=$	$\mathsf{nat} \mid \top \mid \bot \mid A_1 \to A_2 \mid A^{\alpha} \mid \alpha \mid \mu\alpha.\ A \mid A \& B$
Expressions	e	$::=$	$i \mid x \mid e_1\ e_2 \mid \lambda x : A.\ e \mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e$
Values	v	$::=$	$i \mid \lambda x : A.\ e \mid \mathsf{fold}\ [A]\ v$
Contexts	Γ	$::=$	$\cdot \mid \Gamma,\ \alpha \mid \Gamma,\ x : A$

6.5.2 Subtyping

Subtyping is reflexive and transitive, and the unfolding lemma also holds:

Theorem 73 (Reflexivity)**.** If $\vdash \Gamma$ and $\Gamma \vdash A$ then $\Gamma \vdash A \leq A$.

Theorem 74 (Transitivity)**.** If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$.

Lemma 75. Unfolding Lemma.

$$\text{If } \Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B \text{ then } \Gamma \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B.$$

Our size measure approach also works well with intersection types since the size of any premise is strictly less than the size of conclusion among all three rules for intersection types. Not surprisingly, we can extend the measure in Chapter 4.6 by adding

$$height_\Psi(A_1 \& A_2) = max(height_\Psi(A_1),\ height_\Psi(A_2)) + 1$$

Theorem 76 (Decidability of Subtyping)**.** If $\vdash \Gamma$, $\Gamma \vdash A$ and $\Gamma \vdash B$, then $\Gamma \vdash A \leq B$ is decidable.

6.5.3 Typing and reduction

The definition of typing and reduction rules are shown as Figure 6.4. Compared with Figure 3.4, we add the usual typing rule for intersection types (rule TYP-AND): it says if an expression e is of type A and of type B, then it is of type $A\&B$. With the addition to intersection types, a value can have multiple possibilities of types. For example, both $1 : \text{nat}\&\top$ and $\lambda x : ((\text{nat} \to \text{nat})\&(\text{bool} \to \text{bool}))$. x are allowed in our type assignment system.

6.5.4 Type soundness

Compared with the type-safety proof in Chapter 3, after adding intersection types, the proofs are also straightforward, except for some modifications of auxiliary lemmas. All the proofs are mechanized and provided as supplementary material.

Theorem 77 (Preservation)**.** If $\Gamma \vdash e : A$ and $e \rightsquigarrow e'$ then $\Gamma \vdash e' : A$.

Proof. By induction on $\Gamma \vdash e : A$. Most cases are routine except for the case rule TYP-UNFOLD, where we must apply the unfolding lemma (Lemma 75). □

Theorem 78 (Progress)**.** If $\vdash e : A$ then e is a value or $\exists e', e \rightsquigarrow e'$.

6.6 Mechanized Proofs

The folder *coq_record* includes all the Coq proofs about STLC extended with iso-recursive subtyping and record types. The folder *coq_intersection* includes all the Coq proofs about STLC extended with iso-recursive subtyping and record types.

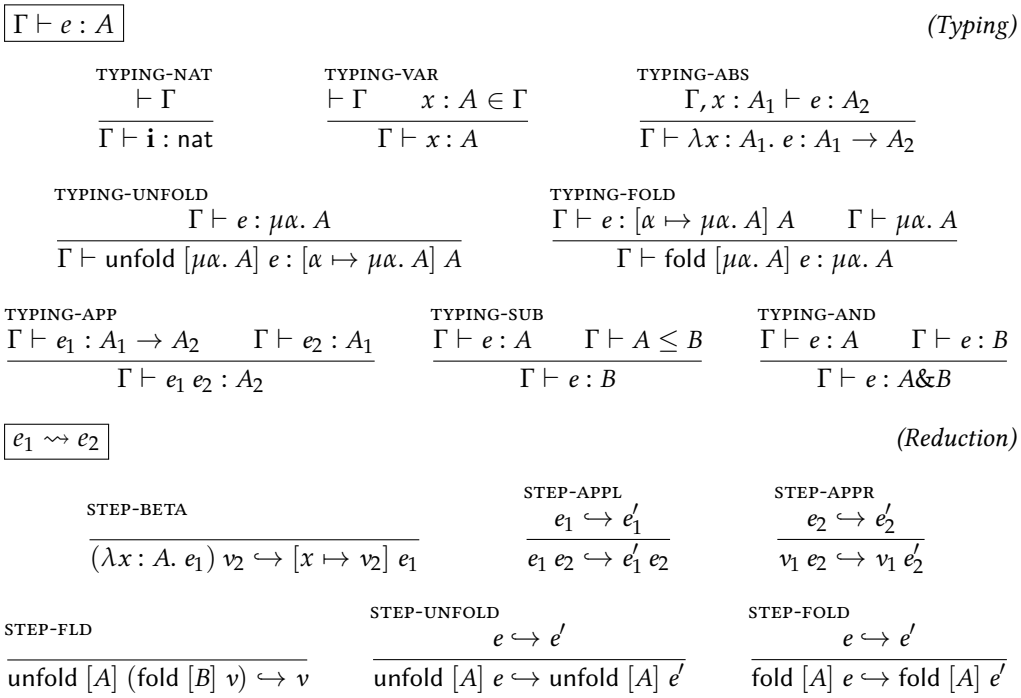


Figure 6.4: Typing and reduction rules (with intersection type).

6.6.1 Definitions

The folder *src_record* includes all the Coq proofs about STLC with iso-recursive subtyping and record types, which corresponds to the calculus in Chapter 6.2. All the definitions in Chapter 6.2 can be found in files *definition.v*.

The folder *src_intersection* includes all the Coq proofs about STLC with iso-recursive subtyping and intersection types, which corresponds to the calculus in Chapter 6.5. All the definitions in Chapter 6.5 can be found in files *definition.v*.

Table 6.1 shows the correspondence of definitions between the paper and the Coq artifacts.

6.6.2 Lemmas and theorems

Table 6.2 shows the descriptions for all the proof scripts in Chapter 6.

Table 6.1: Paper-to-proofs correspondence guide in Chapter 6.

Definition	File	Name in Coq	Notation
in folder *coq_record*			
Well-formed Type (Figure 6.1)	definition.v	WF E A	$\Gamma \vdash A$
Subtyping (Figure 6.1)	definition.v	Sub E A B	$\Gamma \vdash A \leq B$
Typing (Figure 6.2)	definition.v	typing E e A	$\Gamma \vdash e : A$
Reduction (Figure 6.2)	definition.v	step e1 e2	$e_1 \hookrightarrow e_2$
in folder *coq_intersection*			
Well-Formed Type (Figure 6.3)	definition.v	WFS E A	$\Gamma \vdash A$
Subtyping (Figure 6.3)	definition.v	Sub E A B	$\Gamma \vdash A \leq B$
Typing (Figure 6.4)	definition.v	typing E e A	$\Gamma \vdash e : A$
Reduction (Figure 6.4)	definition.v	step e1 e2	$e_1 \rightsquigarrow e_2$

Table 6.2: Descriptions for the proof scripts in Chapter 6.

Theorems	Description	Files	Name in Coq
in folder *coq_record*			
Theorem 65	Reflexivity	subtyping.v	sub_refl
Theorem 66	Transitivity	subtyping.v	Transitivity
Lemma 70	Unfolding Lemma	unfolding.v	unfolding_lemma
Theorem 71	Preservation	typesafety.v	preservation
Theorem 72	Progress	typesafety.v	progress
in folder *coq_intersection*			
Theorem 73	Reflexivity	infra.v	Reflexivity
Theorem 74	Transitivity	infra.v	Transitivity
Theorem 76	Decidability	decidability.v	decidability
Lemma 75	Unfolding Lemma	unfolding.v	unfolding_lemma
Theorem 77	Preservation	typesafety.v	preservation
Theorem 78	Progress	typesafety.v	progress

Chapter 7

A Calculus with the Merge Operator

In Chapter 2.2.1, we introduce the calculi with disjoint intersection types [89]. One calculus supporting nested composition, based on the disjoint intersection types, is called λ_i [20]. In brief, in λ_i, any term constructed via a merge operator has disjoint types or they are the same. For example, both $1,,true$ and $2,,2$ are a valid term while $1,,2$ is invalid. Within λ_i, the typed first-class traits, and other advanced features, are supported [19].

An important limitation of existing calculi with disjoint intersection types is that they lack recursive types. For typed model of objects, supporting recursive types is important, since many object encodings require recursive types [28]. Without recursive types *binary methods* [25] and other types of methods, that refer to the current object type cannot be easily modelled.

In this chapter, a calculus that combines iso-recursive types with disjoint intersection types [89] and a merge operator [106, 57], called λ_i^μ, is presented. The merge operator generalizes symmetric record concatenation, and the calculus supports subtyping as well as recursive types. We use the nominal unfolding rules to add iso-recursive types to a calculus with disjoint intersection types and a merge operator. The main challenge lies in the disjointness definition with iso-recursive subtyping. We show the type soundness of the calculus, decidability of subtyping, as well as the soundness and completeness of our disjointness definition.

7.1 Syntax, Well-Formedness and Subtyping

This section presents the syntax, well-formedness and subtyping of λ_i^μ. We prove *transitivity* of subtyping, the *unfolding lemma* and *decidability* of subtyping.

7.1.1 Syntax and well-formedness

The syntax of our calculus is:

Types	A, B	$::=$	$\mathsf{nat} \mid \top \mid \bot \mid A_1 \to A_2 \mid \{\alpha : A\} \mid \alpha \mid \mu\alpha.\ A \mid A_1 \& A_2$
Expressions	e	$::=$	$i \mid \top \mid x \mid \lambda x.\ e : A \to B \mid e_1\ e_2 \mid \mathsf{fix}\ x : A.\ e \mid e : A$
			$\mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e \mid \{\alpha = e\} \mid e_1 ,, e_2 \mid e.\alpha$
Values	v	$::=$	$i \mid \top \mid \lambda x\ .e : A \to B \mid \mathsf{fold}\ [A]\ v \mid v_1 ,, v_2 \mid \{\alpha = v\}$
Contexts	Γ	$::=$	$\cdot \mid \Gamma,\ \alpha \mid \Gamma,\ x : A$
Modes	$\Leftrightarrow$	$::=$	$\Leftarrow \mid \Rightarrow$

Meta-variables A, B range over types. Types are mostly standard and consist of: natural numbers (nat), the top type ($\top$), the bottom type ($\bot$), function types ($A \to B$), type variables (α), and recursive types ($\mu\alpha.\ A$). The most interesting feature is the presence of new notation of *labelled types* $\{\alpha : A\}$. Here, labelled types are re-denoted as single-field record types, while in Chapter 4.5, the labelled types are denoted as A^{α}. The α in the field of $\{\alpha : A\}$ and the α in the superscript of A^{α} should be considered as the same syntactic constructs, and are different from the recursive variables. We do not change the semantics of labelled types, but we (ab)use such a notation of single-field record types in two different ways: 1) we use them with the nominal unfolding rules for iso-recursive subtyping, as in Chapter 4.5; and 2) we also use them to model records and records types in combination with intersection types and the merge operator. It is also possible to have an alternative design, which should work just as well, that has separate notions of labelled types and records, but such design comes with the cost of a few more subtyping and well-formedness rules.

Expressions, which are denoted as e, include: a top value ($\top$), lambda expressions ($\lambda x.\ e : A \to B$) and fixpoints ($\mathsf{fix}\ x : A.\ e$). Note that for lambda expressions, we annotate both input and output types, since the output types are necessary in a Type-Directed Operational Semantics (TDOS) during reduction, which will be described in Chapter 7.4.1. We also introduce the merge operator ($e_1 ,, e_2$) [106, 57], and annotated expressions ($e : A$).

Values include a canonical top value ($\top$), lambda expressions ($\lambda x.e : A \to B$), merges of values ($v_1 ,, v_2$) and record values ($\{\alpha = v\}$). For proving type-safety, the contexts also store the types of variables used in the program. We employ bi-directional type checking in the system, thus $\Leftarrow/\Rightarrow$ represent the checking mode and synthesis mode, respectively.

The syntactic sugar for record types and records is also shown below, illustrates the standard encoding [57, 106] in terms of intersection types, labelled types and merges.

$$\begin{array}{lcl} \{\alpha_1 : A_1, \ldots, \alpha_n : A_n\} & \equiv & \{\alpha_1 : A_1\} \,\&\, \ldots \,\&\, \{\alpha_n : A_n\} \\ \{\alpha_1 = e_1, \ldots, \alpha_n = e_n\} & \equiv & \{\alpha_1 = e_1\} \,,,\, \ldots \,,,\, \{\alpha_n = e_n\} \end{array}$$

The definition of well-formed types is mostly standard, as Figure 7.1 shows. An environment is well-formed if all the variables are distinct.

7.1.2 Subtyping

Figure 7.2 shows the subtyping relation. Rule S-BOT states that any well-formed type A is a supertype of the $\bot$ type. Rule S-VAR is a standard rule for type variables: variable α is a subtype

$\boxed{\vdash \Gamma}$ *(Well-Formed Environment)*

$$\textsc{wfe-empty}\ \frac{}{\vdash \cdot} \qquad \textsc{wfe-sub}\ \frac{\vdash \Gamma \qquad \alpha \notin \Gamma}{\vdash \Gamma, \alpha} \qquad \textsc{wfe-typ}\ \frac{\vdash \Gamma \qquad x \notin \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, x : A}$$

$\boxed{\Gamma \vdash A}$ *(Well-Formed Type)*

$$\textsc{wft-nat}\ \frac{}{\Gamma \vdash \mathsf{nat}} \qquad \textsc{wft-Top}\ \frac{}{\Gamma \vdash \top} \qquad \textsc{wft-Bot}\ \frac{}{\Gamma \vdash \bot} \qquad \textsc{wft-var}\ \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{wft-arrow}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2}$$

$$\textsc{wft-label}\ \frac{\Gamma \vdash A}{\Gamma \vdash \{\alpha : A\}} \qquad \textsc{wft-rec}\ \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \mu\alpha.\ A} \qquad \textsc{wft-and}\ \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}$$

Figure 7.1: Well-formedness.

of itself. The rule for function types (rule S-ARROW) and intersection types are standard. Rule S-LABEL states that a labelled type is a subtype of another labelled type if the two types are labelled with the same name and $A \leq B$. Rule S-MERGEREC is the nominal unfolding rule. It is the same as the rule SN-REC in Chapter 4.5, except that we adapt the notation of substitution to $[\alpha \mapsto \{\alpha : A\}]\ A$.

A toplike type, whose definition is shown as rule S-TOPLIKE, is both a supertype and a subtype of $\top$. In calculi with disjoint intersection types, the definition of toplike types plays an important role, since disjointness is defined in terms of toplike types. Allowing a larger set of toplike types enables more types to be disjoint. In particular, the motivation for λ_i to include rule TOP-ARROW in subtyping is to allow certain function types to be disjoint [20, 76, 89]. The rule TOP-ARROW itself is inspired from the well-known BCD-subtyping [17] relation, which also states that any function type that returns a toplike type is itself toplike. Rule TOP-ARROW was first adopted in calculi with disjoint intersection types by Bi, Oliveira, and Schrijvers [20], and we follow that approach as well here. Without such rule two function types can never be disjoint, disallowing more than one function in a merge (where all expressions must have disjoint types). Similarly, in λ_i^μ the rule TOP-REC enables merges that can contain more than one expression with a recursive type.

Subtyping is reflexive and transitive:

Theorem 79 (Reflexivity). If $\vdash \Gamma$ and $\Gamma \vdash A$ then $\Gamma \vdash A \leq A$.

Theorem 80 (Transitivity). If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$.

Furthermore, we have also proved the unfolding lemma, which plays an important role in type preservation. The proof strategy is similar to the approach used in Chapter 4.4.

Lemma 81 (Unfolding Lemma). If $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B$ then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B$.

$\boxed{\rceil A \lceil}$ *(Toplike Type)*

$$\textsc{top-base}\ \frac{}{\rceil \top \lceil} \qquad \textsc{top-and}\ \frac{\rceil A \lceil \qquad \rceil B \lceil}{\rceil A \& B \lceil} \qquad \textsc{top-arrow}\ \frac{\rceil B \lceil \qquad \Gamma \vdash A}{\rceil A \to B \lceil} \qquad \textsc{top-rec}\ \frac{\rceil A \lceil}{\rceil \mu\alpha.\ A \lceil} \qquad \textsc{top-rcd}\ \frac{\rceil A \lceil}{\rceil \{\alpha : A\} \lceil}$$

$\boxed{\Gamma \vdash A \leq B}$ *(Subtyping)*

$$\textsc{s-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nat} \leq \mathsf{nat}} \qquad \textsc{s-toplike}\ \frac{\vdash \Gamma \qquad \rceil B \lceil}{\Gamma \vdash A \leq B} \qquad \textsc{s-bot}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash \bot \leq A} \qquad \textsc{s-label}\ \frac{\Gamma \vdash A \leq B}{\Gamma \vdash \{\alpha : A\} \leq \{\alpha : B\}}$$

$$\textsc{s-arrow}\ \frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2} \qquad \textsc{s-and}\ \frac{\Gamma \vdash A \leq B_1 \qquad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \& B_2} \qquad \textsc{s-var}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \alpha}{\Gamma \vdash \alpha \leq \alpha}$$

$$\textsc{s-andR}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \& A_2 \leq B} \qquad \textsc{s-andL}\ \frac{\Gamma \vdash A_2 \qquad \Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \& A_2 \leq B}$$

$$\textsc{s-mergerec}\ \frac{\Gamma, \alpha \vdash [\alpha \mapsto \{\alpha : A\}]\ A \leq [\alpha \mapsto \{\alpha : B\}]\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}$$

Figure 7.2: Subtyping rules.

The proof of decidability is similar to the one used in Theorem 76, except for the cases raised from rule S-TOPLIKE.

Theorem 82 (Decidability of Subtyping)**.** If $\vdash \Gamma$, $\Gamma \vdash A$ and $\Gamma \vdash B$, then $\Gamma \vdash A \leq B$ is decidable.

7.2 Disjointness for Recursive Types

The disjointness restriction is an essential feature in calculi with disjoint intersection types. Such restriction ensures that certain merges of values, that could lead to ambiguity, are forbidden. For instance, in the example above merges are used to encode records. A record $\{x = 1, y = true\}$ is encoded as the merge of two single field records $\{x = 1\},,\{y = true\}$. Here the operator $,,$ is the merge operator [106, 57], which can be viewed as a generalization of record concatenation. Ambiguity can arise with the merge operator if the two values in the merge overlap. For instance, with records, we would like to forbid $r = \{x = 1, x = 2\}$ (a record with two fields with the same name and type), since $r.x$ would be ambiguous. With the merge operator we can merge not only records, but also arbitrary values. Thus, we need to forbid merges such as $1,,2$ which provides two values of type Int. The disjointness restriction is employed when type-checking merges to ensure that the types being merged do not overlap. A standard specification of disjointness [89, 77] is:

Definition 83 (Specification of disjointness). $\Gamma \vdash A *_s B \equiv \forall C, (\Gamma \vdash A \leq C \wedge \Gamma \vdash B \leq C) \Rightarrow \rceil C \lceil$

The intuition is that two types are disjoint when all their supertypes are (isomorphic to) $\top$. The notation $\rceil \cdot \lceil$ represents toplike types, which are both supertypes and subtypes of $\top$. In essence ambiguity arises from upcasts on values. For instance if we cast $1,,2$ under type Int there can be two possible results. Disjointness prevents merges with values having common supertypes (with the exception of $\top$). Therefore, when such disjoint merges are upcast we can ensure that only one value will be extracted for any given (non toplike) type.

One of the challenges in the design of λ_i^μ is to find an algorithmic rule to check whether two recursive types are disjoint and prove that it is complete with respect to the specification. As part of the completeness proof we must be able to find common supertypes of two types, but this is non-trivial for recursive types due to contravariance. For example, assume that we have two recursive types $\mu\alpha.\ ((\mathsf{nat} \to \alpha) \to \mathsf{nat})$ and $\mu\alpha.\ ((\top \to \alpha) \to \mathsf{nat})$, then $\mu\alpha.\ ((\mathsf{nat} \to \alpha)\&(\top \to \alpha) \to \mathsf{nat})$ is not a valid common supertype because α is contravariant. In contrast, for covariant recursive types and non-recursive types, finding a common supertype is simpler. For instance, for the recursive types $\mu\alpha.\ (\mathsf{String} \to \alpha)$ and $\mu\alpha.\ (\mathsf{nat} \to \alpha)$, the intersection of the two inputs types of the function in the recursive type gives us a common supertype $\mu\alpha.\ (\mathsf{String\&nat} \to \alpha)$.

7.2.1 Algorithmic rules of disjointness

Figure 7.4 shows an algorithmic formulation of disjointness. Most rules are standard and follow from previous work [89, 76, 21]. Toplike types are disjoint with other types (rules DIS-TOPL and DIS-TOPR). Intersection types need to check the disjointness of every component (rules DIS-ANDL and DIS-ANDR). Two labelled types are disjoint if they have distinct labels or the types of the label are disjoint (rules DIS-RCDRCD and DIS-RCDRCDEQ). Two different variables are always disjoint (rule DIS-VARVAR). Rule DIS-ARRARR states that, for two function types, we just need to check if their output types are disjoint or not.

The most interesting one is the disjointness of recursive types. Without toplike types, it could be very simple: any two recursive types are not disjoint because $\mu\alpha.\ \top$ is a non toplike common supertype for all recursive types. However, the introduction of toplike types complicates the interaction between any two recursive types. Nevertheless, rule DIS-RECREC is surprisingly simple: two recursive types are disjoint if their bodies are disjoint. Finally, two types with different type constructors (e.g. record types and recursive types) are disjoint (rule DIS-AXIOM).

The soundness lemma showing that our rules satisfy the specification is straightforward:

Lemma 84 (Soundness). If $\Gamma \vdash A * B$ then $\Gamma \vdash A *_s B$.

Proof. By induction on $\Gamma \vdash A * B$. □

$\boxed{\Gamma \vdash A *_{axiom} B}$ *(Disjointness (Axiom))*

DISAM-INTARR
$$\frac{}{\Gamma \vdash \mathsf{nat} *_{axiom} A_1 \to A_2}$$

DISAM-ARRINT
$$\frac{}{\Gamma \vdash A_1 \to A_2 *_{axiom} \mathsf{nat}}$$

DISAM-INTREC
$$\frac{}{\Gamma \vdash \mathsf{nat} *_{axiom} \mu\alpha.\ A}$$

DISAM-RECINT
$$\frac{}{\Gamma \vdash \mu\alpha.\ A *_{axiom} \mathsf{nat}}$$

DISAM-ARRREC
$$\frac{}{\Gamma \vdash A_1 \to A_2 *_{axiom} \mu\alpha.\ A}$$

DISAM-RECARR
$$\frac{}{\Gamma \vdash \mu\alpha.\ A *_{axiom} A_1 \to A_2}$$

DISAM-INTVAR
$$\frac{}{\Gamma \vdash \mathsf{nat} *_{axiom} \alpha}$$

DISAM-VARINT
$$\frac{}{\Gamma \vdash \alpha *_{axiom} \mathsf{nat}}$$

DISAM-ARRVAR
$$\frac{}{\Gamma \vdash A_1 \to A_2 *_{axiom} \alpha}$$

DISAM-VARARR
$$\frac{}{\Gamma \vdash \alpha *_{axiom} A_1 \to A_2}$$

DISAM-RECVAR
$$\frac{}{\Gamma \vdash \mu\alpha.\ A *_{axiom} \beta}$$

DISAM-VARREC
$$\frac{}{\Gamma \vdash \beta *_{axiom} \mu\alpha.\ A}$$

DISAM-RCDINT
$$\frac{}{\Gamma \vdash \{\alpha : A\} *_{axiom} \mathsf{nat}}$$

DISAM-INTRCD
$$\frac{}{\Gamma \vdash \mathsf{nat} *_{axiom} \{\alpha : A\}}$$

DISAM-RCDVAR
$$\frac{}{\Gamma \vdash \{\alpha : A\} *_{axiom} \beta}$$

DISAM-VARRCD
$$\frac{}{\Gamma \vdash \beta *_{axiom} \{\alpha : A\}}$$

DISAM-RCDARR
$$\frac{}{\Gamma \vdash \{\alpha : A\} *_{axiom} A_1 \to A_2}$$

DISAM-ARRRCD
$$\frac{}{\Gamma \vdash A_1 \to A_2 *_{axiom} \{\alpha : A\}}$$

DISAM-RECRCD
$$\frac{}{\Gamma \vdash \mu\alpha.\ A *_{axiom} \{\beta : B\}}$$

DISAM-RCDREC
$$\frac{}{\Gamma \vdash \{\beta : B\} *_{axiom} \mu\alpha.\ A}$$

Figure 7.3: Disjointness (Axiom).

7.2.2 Completeness of disjointness

The most challenging part of the formalization of λ_i^{μ} is to show that algorithmic disjointness is complete with respect to the specification. The difficulty is brought by rule DIS-RECREC. If two recursive types $\mu\alpha.\ A$ and $\mu\alpha.\ B$ satisfy the specification, then for any type C, $\Gamma \vdash \mu\alpha.\ A \leq C \wedge \Gamma \vdash \mu\alpha.\ B \leq C$ implies that C is toplike. By rule DIS-RECREC, we want to prove that any type D satisfying $\Gamma, \alpha \vdash A \leq D \wedge \Gamma, \alpha \vdash B \leq D$ implies that D is toplike. Clearly C and D should be related since in one case C is the supertype of two recursive types, and in the other case D is the supertype of the bodies of the two recursive types. However, the relation between C and D is intricate.

Lower common supertype To help relating C and D, we define a new function $\sqcup$, which is shown in Figure 7.5. The function $\sqcup$ computes a lower supertype of type A and B. A simplification that we employ in our definition is that types of common supertypes in contravariant positions are all $\bot$. Strictly speaking this means that the supertype that we find is not the lowest one in the subtyping lattice. But in our setting this does not matter, because the disjointness of arrow types (see rule DIS-ARRARR) does not account for input types. If the input types did matter for disjointness then we would likely need a dual definition for finding greater common subtypes, making the definition more involved. We can prove some useful properties for $\sqcup$:

Lemma 85 ($\sqcup$ is supertype)**.** For any A and B, $\Gamma \vdash A \leq A \sqcup B$ and $\Gamma \vdash B \leq A \sqcup B$.

$\boxed{\Gamma \vdash A * B}$ *(Disjointness)*

$$\textsc{dis-topL}\ \frac{\rceil B \lceil}{\Gamma \vdash A * B} \qquad \textsc{dis-topR}\ \frac{\rceil A \lceil}{\Gamma \vdash A * B} \qquad \textsc{dis-andL}\ \frac{\Gamma \vdash A_1 * B \qquad \Gamma \vdash A_2 * B}{\Gamma \vdash A_1 \& A_2 * B} \qquad \textsc{dis-andR}\ \frac{\Gamma \vdash A * B_1 \qquad \Gamma \vdash A * B_2}{\Gamma \vdash A * B_1 \& B_2}$$

$$\textsc{dis-VarVar}\ \frac{\alpha \neq \beta}{\Gamma \vdash \alpha * \beta} \qquad \textsc{dis-ArrArr}\ \frac{\Gamma \vdash A_2 * B_2}{\Gamma \vdash A_1 \to A_2 * B_1 \to B_2} \qquad \textsc{dis-RcdRcd}\ \frac{\alpha \neq \beta}{\Gamma \vdash \{\alpha : A\} * \{\beta : B\}}$$

$$\textsc{dis-RcdRcdEq}\ \frac{\Gamma \vdash A * B}{\Gamma \vdash \{\alpha : A\} * \{\beta : B\}} \qquad \textsc{dis-RecRec}\ \frac{\Gamma, \alpha \vdash A * B}{\Gamma \vdash \mu\alpha.\, A * \mu\alpha.\, B} \qquad \textsc{dis-axiom}\ \frac{\Gamma \vdash A *_{axiom} B}{\Gamma \vdash A * B}$$

Figure 7.4: Disjointness.

$$\begin{array}{rclrcl}
\top \sqcup A & = & \top & A \sqcup \top & = & \top \\
\alpha \sqcup \alpha & = & \alpha & \alpha \sqcup \beta & = & \top\ (\alpha \neq \beta) \\
A \sqcup B \& C & = & A \& B \sqcup A \& C & A \& B \sqcup C & = & A \& C \sqcup B \& C \\
\bot \sqcup \mu\alpha.\, A & = & \mu\alpha.\, (\bot \sqcup A) & \mu\alpha.\, A \sqcup \bot & = & \mu\alpha.\, (A \sqcup \bot) \\
\bot \sqcup \{\alpha : A\} & = & \{\alpha : \bot \sqcup A\} & \{\alpha : A\} \sqcup \bot & = & \{\alpha : A \sqcup \bot\} \\
\bot \sqcup A_1 \to A_2 & = & \bot \to (\bot \sqcup A_2) & A_1 \to A_2 \sqcup \bot & = & \bot \to (A_2 \sqcup \bot) \\
\mu\alpha.\, A \sqcup \mu\alpha.\, B & = & \mu\alpha.\, (A \sqcup B) & \{\alpha : A\} \sqcup \{\beta : B\} & = & \top\ (\alpha \neq \beta) \\
A_1 \to A_2 \sqcup B_1 \to B_2 & = & \bot \to (A_2 \sqcup B_2) & \{\alpha : A\} \sqcup \{\alpha : B\} & = & \{\alpha : A \sqcup B\}
\end{array}$$

otherwise: $\bot \sqcup A = A,\ A \sqcup \bot = A,\ A \sqcup A = A,\ A \sqcup B = \top$

Figure 7.5: Lower common supertype.

Lemma 86. If $\Gamma \vdash A \leq C$ and $\Gamma \vdash B \leq C$ and $A \sqcup B$ is toplike, then C is toplike.

Lemma 86 is the most important one: $A \sqcup B$ is not the least common supertype of A and B, but if it is toplike then all supertypes of A and B are toplike. With the previous lemmas we can prove the completeness lemma:

Lemma 87 (Completeness)**.** For types A and B, if $\Gamma \vdash A *_s B$ then $\Gamma \vdash A * B$.

Proof. Do induction on A and B, respectively. All cases are straightforward, except for the case where both types are recursive types. In such case, A is decomposed into $\mu\alpha.\, A'$ and B is decomposed into $\mu\alpha.\, B'$.

1. One of the premises is: $\forall\, C, \Gamma \vdash \mu\alpha.\, A' \leq C \wedge \Gamma \vdash \mu\alpha.\, B' \leq C \Rightarrow \rceil C \lceil$.
2. The induction hypothesis is $(\forall\, D, \Gamma, \alpha \vdash A' \leq D \wedge \Gamma, \alpha \vdash B' \leq D) \Rightarrow \rceil D \lceil \Rightarrow \vdash A' * B'$.
3. The goal is $\Gamma \vdash \mu\alpha.\, A' * \mu\alpha.\, B'$. By applying rule DIS-RECREC and the induction hypothesis, we have two more conditions:
   (a) $\Gamma, \alpha \vdash A' \leq D$;
   (b) $\Gamma, \alpha \vdash B' \leq D$.

   And the goal becomes checking if D is toplike.

4. We could also know two more lemmas from Lemma 85:
   (a) $\Gamma \vdash \mu\alpha.\ A' \leq \mu\alpha.\ A' \sqcup \mu\alpha.\ B'$;
   (b) $\Gamma \vdash \mu\alpha.\ B' \leq \mu\alpha.\ A' \sqcup \mu\alpha.\ B'$.
5. From conditions (1) (4a) and (4b), we get: $\mu\alpha.\ A' \sqcup \mu\alpha.\ B'$ is toplike.
6. According to the definition of lower common supertype for two recursive types, from condition (5), we could know $\mu\alpha.\ (A' \sqcup B')$ is toplike.
7. By inversion of condition (6), we obtain $A' \sqcup B'$ is toplike.
8. Apply Lemma 86 with conditions (3a) and (3b), and we achieve the goal.

□

7.3 Static Semantics of λ_i^μ

We use bidirectional type checking in λ_i^μ, following λ_i [77]. Bi-directional typechecking is helpful to eliminate a source of ambiguity (and non-determinism) that arises from an unrestricted subsumption rule in conventional type assignment systems in the presence of a concatenation/merge operator (a point which was also noted by Cardelli and Mitchell [36]). The typing rules are shown in Figure 7.6. There are two standard modes: $\Gamma \vdash e \Rightarrow A$ synthesises the type A of expression e under the context Γ, and $\Gamma \vdash e \Leftarrow A$ checks if expression e has type A under the context Γ.

Many rules are standard. There are two rules for merge expressions, which follow from previous work by Huang et al. [77]. Rule TYPINGM-MERGE employs a disjointness restriction, and only allows two expressions with disjoint types to be merged. The disjointness restriction prevents ambiguity that could arise merging types with common (non-toplike) supertypes. For instance, if $1,,2$ would be allowed, then in an expression like $(1,,2) + 3$ we could have two possible results: 4 and 5. The merge of duplicated values such as $1,,1$ is not harmful, since no ambiguity arises in this case, and such values can arise from reduction. Thus, there is also a rule TYPINGM-MERGEV, which allows merging two consistent values regardless of their types. The consistency relation is:

Definition 88 (Consistency). $v_1 \approx_{spec} v_2 \equiv \forall A, (v_1 \hookrightarrow_A v_1' \wedge v_2 \hookrightarrow_A v_2') \Rightarrow v_1' = v_2'$

In this relation, two values are consistent if for any type A casting of those two values under type A produces the same result. We introduce the casting relation $v_1 \hookrightarrow_A v_2$, which reduces the value v_1 to v_2 under the type A in Chapter 7.4.1. A key property relating consistency and disjointness is:

Lemma 89 (Consistency of disjoint values). If $\vdash v_1 \Rightarrow A$ and $\vdash v_2 \Rightarrow B$ and $\vdash A *_s B$ then $v_1 \approx_{spec} v_2$.

$\boxed{\Gamma \vdash e \Leftrightarrow A}$ *(Typing)*

TYPINGM-SUB
$$\frac{\Gamma \vdash e \Rightarrow A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e \Leftarrow B}$$

TYPINGM-TOP
$$\frac{\vdash \Gamma}{\Gamma \vdash \top \Rightarrow \top}$$

TYPINGM-VAR
$$\frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A}$$

TYPINGM-APP
$$\frac{\Gamma \vdash e_1 \Rightarrow A_1 \to A_2 \qquad \Gamma \vdash e_2 \Leftarrow A_1}{\Gamma \vdash e_1\, e_2 \Rightarrow A_2}$$

TYPINGM-NAT
$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbf{i} \Rightarrow \mathsf{nat}}$$

TYPINGM-ANNO
$$\frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash e : A \Rightarrow A}$$

TYPINGM-PROJ
$$\frac{\Gamma \vdash e \Rightarrow \{\alpha : A\}}{\Gamma \vdash e.\alpha \Rightarrow A}$$

TYPINGM-RCD
$$\frac{\Gamma \vdash e \Rightarrow A}{\Gamma \vdash \{\alpha = e\} \Rightarrow \{\alpha : A\}}$$

TYPINGM-ABS
$$\frac{\Gamma, x : A_1 \vdash e \Leftarrow A_2}{\Gamma \vdash \lambda x : A_1 \to A_2.\, e \Rightarrow A_1 \to A_2}$$

TYPINGM-UNFOLD
$$\frac{\Gamma \vdash e \Leftarrow \mu\alpha.\, A}{\Gamma \vdash \mathsf{unfold}\ [\mu\alpha.\, A]\ e \Rightarrow [\alpha \mapsto \mu\alpha.\, A]\, A}$$

TYPINGM-FIX
$$\frac{\Gamma, x : A \vdash e \Leftarrow A}{\Gamma \vdash \mathsf{fix}\ x : A.\, e \Rightarrow A}$$

TYPINGM-MERGE
$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma \vdash e_2 \Rightarrow B \qquad \Gamma \vdash A * B}{\Gamma \vdash e_1 ,, e_2 \Rightarrow A \& B}$$

TYPINGM-MERGEV
$$\frac{\cdot \vdash v_1 \Rightarrow A \qquad \cdot \vdash v_2 \Rightarrow B \qquad v_1 \approx_{spec} v_2}{\Gamma \vdash v_1 ,, v_2 \Rightarrow A \& B}$$

TYPINGM-FOLD
$$\frac{\Gamma \vdash e \Leftarrow [\alpha \mapsto \mu\alpha.\, A]\, A \qquad \Gamma \vdash \mu\alpha.\, A}{\Gamma \vdash \mathsf{fold}\ [\mu\alpha.\, A]\ e \Rightarrow \mu\alpha.\, A}$$

Figure 7.6: Typing rules for λ_i^μ.

7.4 Dynamic Semantics of λ_i^μ

We now introduce the Type-Directed Operational Semantics (TDOS) for λ_i^μ. TDOS, originally proposed by Huang and Oliveira [76] and Huang et al. [77], is a variant of small-step operational semantics. In TDOS, type annotations are operationally relevant, since selecting values from merged values is type-directed. We show that λ_i^μ is deterministic and type sound.

7.4.1 A Type-Directed Operational Semantics for λ_i^μ

The defining feature of a TDOS is a relation called casting (originally called typed reduction by Huang et al.). Casting plays an important role: based on the contextual type information, values are further reduced to match the type structure precisely. In many conventional operational semantics a value is the final result in a program, but with TDOS further reduction can happen if the type that is required for the value has a mismatch with the shape of the value. For example, if we have the merge $1 ,, ' c'$ at type Int then casting will produce 1. However, the same value at type $Int \& Char$ would remain unchanged ($1 ,, ' c'$).

The rules for casting are shown at the Figure 7.7. All non-intersection types are ordinary types. Casting $v_1 \hookrightarrow_A v_2$ denotes that the value v_1 is reduced to v_2 under the type A. From the definitions, we can see that the A is the supertype of the principal type of v_1, and v_2 is the

$\boxed{\textbf{ord}\ A}$ *(Ordinary Type)*

$$\textsc{ord-nat}\ \frac{}{\textbf{ord}\ \mathsf{nat}} \qquad \textsc{ord-top}\ \frac{}{\textbf{ord}\ \top} \qquad \textsc{ord-bot}\ \frac{}{\textbf{ord}\ \bot} \qquad \textsc{ord-arr}\ \frac{}{\textbf{ord}\ A \to B} \qquad \textsc{ord-rec}\ \frac{}{\textbf{ord}\ \mu\alpha.\ A} \qquad \textsc{ord-var}\ \frac{}{\textbf{ord}\ \alpha}$$

$$\textsc{ord-rcd}\ \frac{}{\textbf{ord}\ \{l : A\}}$$

$\boxed{v_1 \hookrightarrow_A v_2}$ *(Casting)*

$$\textsc{tred-nat}\ \frac{}{\mathbf{i} \hookrightarrow_{\mathsf{nat}} \mathbf{i}} \qquad \textsc{tred-top}\ \frac{\textbf{ord}\ A \qquad \rceil A \lceil}{v \hookrightarrow_A A^\dagger} \qquad \textsc{tred-mergeL}\ \frac{v_1 \hookrightarrow_A v \qquad \textbf{ord}\ A}{v_1 ,, v_2 \hookrightarrow_A v} \qquad \textsc{tred-mergeR}\ \frac{v_2 \hookrightarrow_A v \qquad \textbf{ord}\ A}{v_1 ,, v_2 \hookrightarrow_A v}$$

$$\textsc{tred-rec}\ \frac{\sim\rceil \mu\alpha.\ B \lceil \qquad \cdot \vdash \mu\alpha.\ A \leq \mu\alpha.\ B}{\mathsf{fold}\ [\mu\alpha.\ A]\ v \hookrightarrow_{\mu\alpha.\ B} \mathsf{fold}\ [\mu\alpha.\ B]\ v} \qquad \textsc{tred-arrow}\ \frac{\sim\rceil B_2 \lceil \qquad \cdot \vdash B_1 \leq A_1 \qquad \cdot \vdash A_2 \leq B_2}{\lambda x : A_1 \to A_2.\ e \hookrightarrow_{B_1 \to B_2} \lambda x : A_1 \to B_2.\ e}$$

$$\textsc{tred-and}\ \frac{v \hookrightarrow_A v_1 \qquad v \hookrightarrow_B v_2}{v \hookrightarrow_{A\&B} v_1 ,, v_2} \qquad \textsc{tred-rcd}\ \frac{v_1 \hookrightarrow_A v_2 \qquad \sim\rceil \{l : A\} \lceil}{\{\alpha = v_1\} \hookrightarrow_{\{l:A\}} \{\alpha = v_2\}}$$

Figure 7.7: Casting.

value compatible with A. The most special one is rule TRED-TOP: if the type is toplike, then a value will reduce to the corresponding top value, where the $A^\dagger$ is defined as:

$$\begin{array}{rclcrcl} (A \to B)^\dagger & = & \lambda x\ .\top : A \to B & \qquad & (\mu\alpha.\ A)^\dagger & = & \mathsf{fold}\ [\mu\alpha.\ A]\ \top \\ \{\alpha : A\}^\dagger & = & \{\alpha : A^\dagger\} & & (A\&B)^\dagger & = & A^\dagger ,, B^\dagger \end{array}$$

$$\text{otherwise:} A^\dagger = \top$$

7.4.2 Reduction

The definition of reduction is shown at the Figure 7.8. Most rules are standard. Casting is used in rule STEP-BETAN for adjusting the argument value to the expected type for the input of the function. Casting is also used in rule STEP-ANNOV for annotations. Rules STEP-FLDN and STEP-FLDT are for unfold expressions. Finally, there is also a special rule STEP-FLDM for recursive types as well as intersection types.

7.4.3 Determinism

One of the properties of our semantics is determinism: expressions will always reduce to the same value. Lemma 90 says that if a value can be type-checked, then it reduces to a same value under the type A. Lemma 91 says that if an expression can be type-checked, then it reduces to a unique expression.

Lemma 90 (Determinism of $\hookrightarrow_A$)**.** If $\Gamma \vdash v \Rightarrow B$ and $v \hookrightarrow_A v_1$ and $v \hookrightarrow_A v_2$ then $v_1 = v_2$.

Proof. By induction on $v \hookrightarrow_A v_1$. □

$\boxed{e_1 \hookrightarrow e_2}$ *(Reduction)*

$$\textsc{step-betan}\ \frac{v_1 \hookrightarrow_{A_1} v_2}{(\lambda x : A_1 \to A_2.\ e)\ v_1 \hookrightarrow ([x \mapsto v_2]\ e) :\ A_2} \qquad \textsc{step-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2} \qquad \textsc{step-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\ e_2 \hookrightarrow v_1\ e_2'}$$

$$\textsc{step-proj}\ \frac{e \hookrightarrow e'}{e.l_j \hookrightarrow e'.l_j} \qquad \textsc{step-mergel}\ \frac{e_1 \hookrightarrow e_1'}{e_1,, e_2 \hookrightarrow e_1',, e_2} \qquad \textsc{step-merger}\ \frac{e_2 \hookrightarrow e_2'}{v_1,, e_2 \hookrightarrow v_1,, e_2'} \qquad \textsc{step-anno}\ \frac{e \hookrightarrow e'}{e :\ A \hookrightarrow e' :\ A}$$

$$\textsc{step-annov}\ \frac{v \hookrightarrow_A v'}{v :\ A \hookrightarrow v'} \qquad \textsc{step-fix}\ \frac{}{\mathsf{fix}\ x : A.\ e \hookrightarrow ([x \mapsto \mathsf{fix}\ x : A.\ e]\ e) :\ A} \qquad \textsc{step-fold}\ \frac{e \hookrightarrow e'}{\mathsf{fold}\ [A]\ e \hookrightarrow \mathsf{fold}\ [A]\ e'}$$

$$\textsc{step-label}\ \frac{e \hookrightarrow e'}{\{\alpha = e\} \hookrightarrow \{\alpha = e'\}} \qquad \textsc{step-unfold}\ \frac{e \hookrightarrow e'}{\mathsf{unfold}\ [A]\ e \hookrightarrow \mathsf{unfold}\ [A]\ e'}$$

$$\textsc{step-fldn}\ \frac{v_1 \hookrightarrow_{[\alpha \mapsto \mu\alpha.\ A]\ A} v_2 \qquad \sim\rceil \mu\alpha.\ A \lceil}{\mathsf{unfold}\ [\mu\alpha.\ A]\ (\mathsf{fold}\ [\mu\alpha.\ B]\ v_1) \hookrightarrow v_2} \qquad \textsc{step-fldm}\ \frac{v_1,, v_2 \hookrightarrow_{\mu\alpha.\ A} v \qquad \sim\rceil \mu\alpha.\ A \lceil}{\mathsf{unfold}\ [\mu\alpha.\ A]\ (v_1,, v_2) \hookrightarrow \mathsf{unfold}\ [\mu\alpha.\ A]\ v}$$

$$\textsc{step-fldt}\ \frac{v_1 \hookrightarrow_{[\alpha \mapsto \mu\alpha.\ A]\ A} v_2 \qquad \rceil \mu\alpha.\ A \lceil}{\mathsf{unfold}\ [\mu\alpha.\ A]\ v_1 \hookrightarrow v_2} \qquad \textsc{step-projlabel}\ \frac{}{\{\alpha = v\}.\alpha \hookrightarrow v}$$

Figure 7.8: Small-step semantics.

Lemma 91 (Determinism of $\hookrightarrow$)**.** If $\Gamma \vdash e \hookrightarrow A$ and $e \hookrightarrow e_1$ and $e \hookrightarrow e_2$ then $e_1 = e_2$.

Proof. By induction on $e \hookrightarrow e_1$. □

7.4.4 Type safety

We prove type safety following a similar approach to the previous work by Huang, Zhao, and Oliveira [77], and by showing progress and preservation theorems. The following lemmas and theorems show that our system is type-safe.

Theorem 92 (Preservation)**.** If $\vdash e_1 \Leftrightarrow A$ and $e_1 \hookrightarrow e_2$ then $\vdash e_2 \Leftarrow A$.

Proof. By induction on $\vdash e_1 \Leftrightarrow A$. □

Lemma 93 (Progress of $\hookrightarrow_A$)**.** If $\vdash v_1 \Leftarrow A$ then $\exists v_2,\ v_1 \hookrightarrow_A v_2$.

Proof. By induction on A. □

Theorem 94 (Progress)**.** If $\vdash e_1 \Leftrightarrow A$ then e_1 is a value or $\exists e_2,\ e_1 \hookrightarrow e_2$.

Proof. By induction on $\vdash e_1 \Leftrightarrow A$. □

Table 7.1: Paper-to-proofs correspondence guide in Chapter 7.

Definition	File	Name in Coq	Notation
Well-formed Type (Figure 7.1)	def_typ.v	WFS E A	$\Gamma \vdash A$
Toplike type (Figure 7.2)	def_typ.v	toplike E A	$\rceil A \lceil$
Subtyping (Figure 7.2)	def_typ.v	Sub E A B	$\Gamma \vdash A \leq B$
Disjointness (Figure 7.4)	def_typ.v	dis E A B	$\Gamma \vdash A * B$
Disjointness (Definition 83)	disjoint.v	disjointSpec E A B	$\Gamma \vdash A *_s B$
Lower common supertype (Figure 7.5)	lub.v	Lub E A B	$A \sqcup B$
Typing (Figure 7.6)	def_exp.v	typing E e A	$\Gamma \vdash e : A$
Casting (Figure 7.7)	def_exp.v	typ_reduce e1 A e2	$e_1 \hookrightarrow_A e_2$
Reduction (Figure 7.8)	def_exp.v	step e1 e2	$e_1 \hookrightarrow e_2$

Table 7.2: Descriptions for the proof scripts in Chapter 7.

Theorems	Description	Files	Name in Coq
Theorem 79	Reflexivity	infra.v	Reflexivity
Theorem 80	Transitivity	infra.v	Transitivity
Theorem 82	Decidability	decidability.v	decidability
Lemma 81	Unfolding Lemma	unfolding.v	unfolding_lemma
Lemma 84	Soundness of disjointness	disjoint.v	disjoint_soundness
Lemma 87	Completeness of disjointness	disjoint.v	disjoint_completeness
Lemma 90	Determinism of Casting	typereduce.v	TypedReduce_unique
Lemma 91	Determinism of Reduction	progress.v	step_unique
Theorem 92	Preservation	typesafety.v	preservation
Lemma 93	Progress of Casting	progress.v	TypedReduce_progress
Theorem 94	Progress	typesafety.v	progress

7.5 Mechanized Proofs

The folder *coq_merge* includes all the Coq proofs about STLC extended with iso-recursive subtyping, intersection types and merge operator, which is the calculus described in this chapter.

7.5.1 Definitions

Table 7.1 shows the correspondence of definitions between the paper and the Coq artifacts. File *def_typ.v* contains definitions regarding types and some infrastructure lemmas for the locally nameless representation. Note that in the coq proof, the toplike relation is a pair of a context and a type, which is different from the paper. In the paper, for readability, the context is omitted, instead, we use the notation $\rceil A \lceil$ to denote the relation. File *lub.v* contains definitions of lower common supertypes and lemmas regarding them. File *def_exp.v* contains definitions regarding expressions and some infrastructure lemmas for the locally nameless representation.

7.5.2 Lemmas and theorems

Table 7.2 shows the descriptions for all the proof scripts in Chapter 7.

Chapter 8

Calculus with Bounded Quantification

In this chapter an extension of kernel $F_{\leq}$, called $F^{\mu}_{\leq}$, with iso-recursive types, is presented. $F_{\leq}$ is a well-known polymorphic calculus with bounded quantification. In $F^{\mu}_{\leq}$ we add iso-recursive types, and correspondingly extend the subtyping relation with iso-recursive subtyping using the recently proposed nominal unfolding rules. We also add two smaller extensions to $F_{\leq}$. The first one is a generalization of the kernel $F_{\leq}$ rule for bounded quantification that accepts *equivalent* rather than *equal* bounds. The second extension is the use of so-called *structural* folding/unfolding rules, inspired by the structural unfolding rule proposed by Abadi et al. [3]. The structural rules add expressive power to the more conventional folding/unfolding rules in the literature, and they enable additional applications. We present several results, including: type soundness; transitivity and decidability of subtyping; the conservativity of $F^{\mu}_{\leq}$ over $F_{\leq}$; and a sound and complete algorithmic formulation of $F^{\mu}_{\leq}$. Moreover, we study an extension of $F^{\mu}_{\leq}$ with both top and bottom types, and both upper and lower bounds instead of the conventional (upper) bounded quantification of $F_{\leq}$.

8.1 Overview

Both bounded quantification and recursive types are the prominent features in many modern programming languages, such as Java, Scala or TypeScript. The classic application for both features is encodings of objects [28]. In addition, the two features are useful to model encodings of algebraic datatypes with subtyping.

8.1.1 Object encodings

A simple and well-known typed encoding of objects is the recursive records encoding [28, 29, 49]. In this encoding the idea is that object types are encoded as recursive record types, and objects are encoded as records. We will use a simplified form of the encoding, where we do not deal with self-references. But self-references could be dealt with in standard ways. For

example, we can define a type Point:

$$\mathsf{Point} \triangleq \mu\ \mathsf{pnt}.\{\mathsf{x} : \mathsf{Int},\ \mathsf{y} : \mathsf{Int},\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{pnt}\}$$

which consists of its coordinates and a move function. We use a recursive type because move should return an updated point. To implement Point we define some auxiliary functions:

```
function getX(p : Point) = (unfold [Point] p).x
function getY(p : Point) = (unfold [Point] p).y
function moveTo(p : Point, x : Int, y : Int) = (unfold [Point] p)
  .move x y
```

then a constructor mkPoint can be defined as:

```
function mkPoint(x1 : Int, y1 : Int) = fold [Point] { x=x1, y=y1,
  move = λx2 y2. mkPoint(x2, y2) }
```

Note that the auxiliary functions above would not be needed in a source language, since a source language would treat p.x as syntactic sugar for (unfold [Point] p).x. Similarly, the source language would automatically insert a fold in the object constructor. In other words, in a source language with iso-recursive subtyping the fold's and unfold's do not need to be explicitly written and are automatically inserted by the compiler. For instance, this is what Abadi, Cardelli, and Viswanathan [3]'s translation of a language with objects into an iso-recursive extension of $F_{\leq}$ does.

With subtyping, we can develop subtypes of Point, such as:

$$\mathsf{ColorPoint} \triangleq \mu\ \mathsf{pnt}.\{\mathsf{x} : \mathsf{Int},\ \mathsf{y} : \mathsf{Int},\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{pnt},\ \mathsf{color} : \mathsf{String}\}$$
$$\mathsf{EqPoint} \triangleq \mu\ \mathsf{pnt}.\{\mathsf{x} : \mathsf{Int},\ \mathsf{y} : \mathsf{Int},\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{pnt},\ \mathsf{eq} : \mathsf{pnt} \to \mathsf{Bool}\}$$

Now we wish to translate the coordinates by one unit for a point, but we do not want to write such a translation function for all subclasses of Point. This is achieved with a polymorphic function:

```
function translate [P ≤ Point] (p : P) = (unfold [Point] p).move
   (getX p + 1) (getY p + 1)
```

The type of this translate function is $\forall(\mathsf{P} \leq \mathsf{Point}).\ \mathsf{P} \to \mathsf{Point}$, which is obtained from the following typing derivation (some parts ommited):

$$\dots\ \frac{\dfrac{\dfrac{\mathsf{P} \leq \mathsf{Point}, \mathsf{p} : \mathsf{P} \vdash \mathsf{p} : \mathsf{P} \qquad \mathsf{P} \leq \mathsf{Point}, \mathsf{p} : \mathsf{P} \vdash \mathsf{P} \leq \mathsf{Point}}{\mathsf{P} \leq \mathsf{Point}, \mathsf{p} : \mathsf{P} \vdash \mathsf{p} : \mathsf{Point}}\ \textsc{typing-sub}}{\mathsf{P} \leq \mathsf{Point}, \mathsf{p} : \mathsf{P} \vdash (\mathsf{unfold}\ [\mathsf{Point}]\ \mathsf{p}) : \left\{\begin{array}{l}\mathsf{x} : \mathsf{Int},\ \mathsf{y} : \mathsf{Int},\\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Point}\end{array}\right\}}\ \textsc{typing-unfold}}{\vdash \mathsf{translate} : \forall(\mathsf{P} \leq \mathsf{Point}).\ \mathsf{P} \to \mathsf{Point}}$$

In the derivation, the conventional unfolding rule is used. Recall the traditional typing rules for folding and unfolding recursive types:

$$\textsc{typing-unfold}\ \frac{\Gamma \vdash e : \mu\alpha.\ A}{\Gamma \vdash \mathsf{unfold}\ [\mu\alpha.\ A]\ e : [\alpha \mapsto \mu\alpha.\ A]\ A} \qquad \textsc{typing-unfold}\ \frac{\Gamma \vdash e : \mu\alpha.\ A}{\Gamma \vdash \mathsf{unfold}\ [\mu\alpha.\ A]\ e : [\alpha \mapsto \mu\alpha.\ A]\ A}$$

In both the rule TYPING-UNFOLD and rule TYPING-FOLD the annotation *must* be a recursive type. However, this type is unsatisfying because it loses type precision: it returns a Point instead of a P. The type that we want instead is:

$$\forall(\mathsf{P} \leq \mathsf{Point}).\ \mathsf{P} \to \mathsf{P}$$

Unfortunately, it is impossible to obtain this more general type with only bounded quantification, and the conventional unfolding rule TYPING-UNFOLD. If we wish to return P, then we should be able to use unfold with the annotation P, which is not a recursive type, but rather a type variable. Some advanced techniques, such as *f-bounded quantification* [29], have been proposed to address this issue. $F^{\mu}_{\leq}$ uses a less intrusive approach to achieve the desired typing for translate, namely the structural rules, which contains

$$\textsc{typing-sunfold}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq \mu\alpha.\ B}{\Gamma \vdash \mathsf{unfold}\ [A]\ e : [\alpha \mapsto A]\ B} \qquad \textsc{typing-sunfold}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq \mu\alpha.\ B}{\Gamma \vdash \mathsf{unfold}\ [A]\ e : [\alpha \mapsto A]\ B}$$

The key point about the structural rules is that the annotations are generalized to be a *subtype-/supertype* of a recursive type, instead of exactly a recursive type. In particular, this generalization enables annotating fold/unfold with a bounded type variable, which is a subtype/supertype of a recursive type. This is forbidden in the traditional rules. In the rule TYPING-SUNFOLD, it is worthwhile to mention that when we have $A \leq \mu\alpha.\ B$ where α appears negatively in B, then there are very limited choices to what A can be. Essentially it can be $\mu\alpha.\ B$ itself and little else. In other words, negative recursive types have very restricted subtyping, which is why the structural unfolding rule can be type safe. Note also that, since the structural unfolding rules provide almost no flexibility for negative recursive subtyping, they are insufficient to fully express f-bounded quantification for negative recursive types.

8.1.2 Encoding positive f-bounded quantification

Fortunately, with the structural rules, we are allowed to use a type variable as an annotation for unfold. This enables us to encode forms of f-bounded quantification with positive occurrences of recursive variables, which is the case for the Point recursive type. For example, we can change the unfold annotation in translate from the recursive type Point to its subtype, the bounded universal type variable P, as follows:

```
function translate [P ≤ Point] (p : P) = (unfold [P] p).move (
  getX p + 1) (getY p + 1)
```

After this change the type of translate is $\forall(\mathsf{P} \leq \mathsf{Point}).\ \mathsf{P} \to \mathsf{P}$, as the derivation shows:

$$\dots \frac{\text{TYPING-SUNFOLD}\ \dfrac{\mathsf{P} \leq \mathsf{Point}, \mathsf{p} : \mathsf{P} \vdash \mathsf{p} : \mathsf{P} \qquad \mathsf{P} \leq \mathsf{Point}, \mathsf{p} : \mathsf{P} \vdash \mathsf{P} \leq \mathsf{Point}}{\mathsf{P} \leq \mathsf{Point}, \mathsf{p} : \mathsf{P} \vdash (\mathsf{unfold}\ [\mathsf{P}]\ \mathsf{p}) : \left\{ \begin{array}{l} \mathsf{x} : \mathsf{Int},\ \mathsf{y} : \mathsf{Int}, \\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{P} \end{array} \right\}}}{\vdash \mathsf{translate} : \forall(\mathsf{P} \leq \mathsf{Point}).\ \mathsf{P} \to \mathsf{P}}$$

Then we can apply translate to Point or any of its subtypes, without losing static precision. Thus, if we call translate [EqPoint] (mkEqPoint 0 0), then we obtain an EqPoint object at $(1, 1)$. Note that here mkEqPoint is a constructor for objects with type EqPoint, which contain a binary method [25] eq.

```
function mkEqPoint(x1 : Int, y1 : Int) = fold [EqPoint]
  { x = x1, y = y1, move = λx2 y2. mkEqPoint(x2, y2), eq = λp. (
  getX p == x1) ∧ (getY p == y1) }
```

8.1.3 Encodings of algebraic datatypes

It is well-known that in the polymorphic lambda calculus (System F) [67, 107], we can use Church encodings [45] to encode algebraic datatypes [22]. However, Church encodings make it hard to encode some operations, or worst they can prevent the encoding of certain operations with the correct time complexity. A well-known example of this, due to Church himself, is the encoding of the predecessor function on natural numbers, which has to be a linear time operation with Church encodings instead of a constant time operation.

An alternative encoding of datatypes in the untyped lambda calculus, which avoids the issues of Church encodings, is due to Scott [110]. Unfortunately, Scott encodings cannot be encoded in plain System F. However, the addition of recursive types to a polymorphic lambda calculus allows a typed encoding for Scott encodings [93]. Moreover, in the presence of subtyping, we can also encode algebraic datatypes with subtyping, enabling certain forms of reuse that are not possible without subtyping. Oliveira [90] has shown this assuming a $F_{\leq}$-like language with recursive types, records and higher kinds, but he has not formalized such a language. Here we revisit Oliveira's example. A similar encoding for datatypes can be achieved in $F^{\mu}_{\leq}$. For example, one may define a datatype Exp_1 for mathematical expressions, with numeric, addition, and subtraction constructors:

```
data Exp1 = Num Int | Add Exp1 Exp1 | Sub Exp1 Exp1
```

The encoding in $F^{\mu}_{\leq}$ of this datatype can be defined as follows:

$$\mathsf{Exp}_1 \triangleq \mu\mathsf{E}.\ \forall\mathsf{A}.\ \{\mathsf{num} : \mathsf{Int} \to \mathsf{A},\ \mathsf{add} : \mathsf{E} \to \mathsf{E} \to \mathsf{A},\ \mathsf{sub} : \mathsf{E} \to \mathsf{E} \to \mathsf{A}\} \to \mathsf{A}$$

If we unfold the recursive type, this encoding is indeed a polymorphic higher order function and takes a record with three fields (num, add and sub) as input. Each field corresponds to a constructor in the datatype definition. This encoding is particularly useful for case analysis, since the polymorphic function essentially encodes case analysis directly. To write a function that performs case analysis on this datatype, one can unfold the recursive type, instantiate A with the result type, and then provide a record that maps each case to its implementation

function that takes the constructor components as input and returns a result of type A. For example, given a datatype instance e typed Exp_1, a case analysis-based evaluation function can be written as:

```
function eval (e : Exp₁) = (unfold [Exp₁] e) [Int]
  { num = λn. n,  add = λe₁ e₂. eval e₁ + eval e₂,  sub = λe₁ e₂.
  eval e₁ - eval e₂ }
```

where we use $[\ldots]$ to represent type instantiation. Here Exp_1 is instantiated with the evaluation result type Int. A record of three functions is supplied to implement case analysis. The num field implements a function that returns the integer component n of Num constructor directly, while the functions in add and sub fields perform the evaluation process recursively.

To construct concrete instances of the datatype, each constructor also comes with a corresponding encoding in the calculus:

```
function Num₁ (n: Int) = fold [Exp₁] (Λ A. λ e. (e.num n))
function Add₁ (e₁ : Exp₁, e₂ : Exp₁) = fold [Exp₁] (Λ A. λ e. (e.
  add e₁ e₂))
function Sub₁ (e₁ : Exp₁, e₂ : Exp₁) = fold [Exp₁] (Λ A. λ e. (e.
  sub e₁ e₂))
```

One can easily check, using rule TYPING-FOLD, that the result type of each constructor encoding becomes Exp_1 after a recursive type folding. Therefore, in this encoding, the use of constructors and case analysis functions is natural: one can construct the expression $1 + 2$ directly with the encoded constructors as $\mathsf{Add}_1\ (\mathsf{Num}_1\ 1)\ (\mathsf{Num}_1\ 2)$, and get its evaluation result by calling eval $(\mathsf{Add}_1\ (\mathsf{Num}_1\ 1)\ (\mathsf{Num}_1\ 2))$.

8.1.4 Subtyping between datatypes

Now consider a larger datatype Exp_2, which extends the Exp_1 datatype with a new constructor Neg, for denoting negative numbers.

```
data Exp₂ = Num Int | Add Exp₂ Exp₂ | Sub Exp₂ Exp₂ | Neg Exp₂
```

This datatype is encoded in $F^{\mu}_{\leq}$ as:

$$\mathsf{Exp}_2 \triangleq \mu\mathsf{E}.\ \forall\mathsf{A}.\ \{\mathsf{num} : \mathsf{Int} \to \mathsf{A},\ \mathsf{add} : \mathsf{E} \to \mathsf{E} \to \mathsf{A},\ \mathsf{sub} : \mathsf{E} \to \mathsf{E} \to \mathsf{A}, \mathsf{neg} : \mathsf{E} \to \mathsf{A}\} \to \mathsf{A}$$

The datatype Exp_2 differs from Exp_1 only in the new constructor. However, other constructors are just the same. To reduce code duplication, polymorphism on datatype constructors is desirable. Note that Exp_2 has more constructors than Exp_1, so it should be safe to coerce Exp_1 expressions into Exp_2 expressions, i.e. $\mathsf{Exp}_1 \leq \mathsf{Exp}_2$. Therefore, we would like the constructor for Add to have the following type, so that both Exp_1 and Exp_2 can use this constructor:

$$\mathsf{Add}_{\forall} : \forall(\mathsf{E} \geq \mathsf{Exp}_1).\ \mathsf{E} \to \mathsf{E} \to \mathsf{E}$$

There are two problems here. Firstly, similarly to the issue that we have faced in the translate function, we would like to use a type variable in the fold's of the constructors. This way we can make the constructors polymorphic. Secondly, as evidenced by the desired type for Add, we need *lower bounded quantification*, but in $F^{\mu}_{\leq}$ (and $F_{\leq}$) we only have upper bounded quantification.

8.1.5 Polymorphic constructors with lower bounded quantification

For applications such as encodings of algebraic datatypes, the dual form of bounded quantification (lower bounded quantification) seems to be more useful. Thus we have an extended system, called $F^{\mu}_{\leq\geq}$, that has both upper and lower bounded quantification. Polymorphic datatype constructors become typeable with the structural folding rule. For example, we can encode the polymorphic Add constructor as:

```
function Add∀ [E ≥ Exp1] (e1 : E, e2 : E) = fold [E] (Λ A. λ e. (
  e.add e1 e2))
```

With lower bounded quantification and the structural folding rules we can get the correct typing for the polymorphic Add constructor:

$$\text{TYPING-SFOLD}\ \frac{\dots \qquad \dots}{\dots\ \dfrac{\mathsf{E} \geq \mathsf{Exp}_1, \mathsf{e}_1 : \mathsf{E}, \mathsf{e}_2 : \mathsf{E} \vdash \Lambda\mathsf{A}.\, \lambda\mathsf{e}.\, (\mathsf{e.add}\ \mathsf{e}_1\ \mathsf{e}_2) : \forall\mathsf{A}. \left\{ \begin{array}{l} \mathsf{num} : \mathsf{Int} \to \mathsf{A}, \\ \mathsf{add} : \mathsf{E} \to \mathsf{E} \to \mathsf{A}, \\ \mathsf{sub} : \mathsf{E} \to \mathsf{E} \to \mathsf{A} \end{array} \right\} \to \mathsf{A}}{\mathsf{E} \geq \mathsf{Exp}_1, \mathsf{e}_1 : \mathsf{E}, \mathsf{e}_2 : \mathsf{E} \vdash \mathsf{fold}\ [\mathsf{E}]\ (\Lambda\mathsf{A}.\, \lambda\mathsf{e}.\, (\mathsf{e.add}\ \mathsf{e}_1\ \mathsf{e}_2)) : \mathsf{E}}}{\vdash \mathsf{Add}_\forall : \forall(\mathsf{E} \geq \mathsf{Exp}_1).\, \mathsf{E} \to \mathsf{E} \to \mathsf{E}}$$

Other polymorphic constructors such as $\mathsf{Num}_\forall$ and $\mathsf{Sub}_\forall$ can be encoded similarly.

With polymorphic datatype constructors, more useful programming patterns can be further exploited. For example, if we want to implement a compiler that uses Exp_1 as its core language, but also want to support richer datatype constructors in a source language like Exp_2 does, we would like to be able to reduce code duplication across the two similar languages. For instance, if we define a pretty printer function for Exp_2:

```
function print (e: Exp2) = (unfold [Exp2] e) [string] {
  num = λ n. (int_to_string n),
  add = λ e1 e2. ((print e1) ++ "+" ++ (print e2)),
  sub = λ e1 e2. ((print e1) ++ "-" ++ (print e2)),
  neg = λ e. ("-" ++ (print e))
}
```

It should be natural to use this function to print Exp_1 expressions as well, since all the constructors in Exp_1 are also in Exp_2 and have their pretty printing methods defined in the above function. In fact, with subtyping between algebraic datatypes, it holds that $\mathsf{Exp}_1 \leq \mathsf{Exp}_2$, so it is safe in our encodings to apply this print function to values of type Exp_1, without requiring another pretty printing function for Exp_1.

Suppose that we wish to implement a simple desugaring function that transforms Exp_2 into Exp_1, by transforming negative numbers $-n$ into subtractions $0 - n$. This function should do case analysis on Exp_2 and use *only* the constructors in Exp_1 to produce the result, i.e. it should have a type $\mathsf{Exp}_2 \to \mathsf{Exp}_1$. The following code can have the desired precise typing naturally with polymorphic datatype constructors:

```
function desugar (e: Exp₂) = (unfold [Exp₂] e) [Exp₁] {
  num = λ n. Num∀ [Exp₁] n,
  add = λ e₁ e₂. Add∀ [Exp₁] (desugar e₁) (desugar e₂),
  sub = λ e₁ e₂. Sub∀ [Exp₁] (desugar e₁) (desugar e₂),
  neg = λ e. Sub∀ [Exp₁] (Num∀ [Exp₁] 0) (desugar e)
}
```

In contrast, in many practical programming languages this task either involves code duplication or loss of type precision. In a typical functional language, we can define both Exp_1 and Exp_2 and also obtain precise static typing guarantees for the desugar function. But this comes at the cost of duplication, since the constructors for the two datatypes are different, and many operations, such as pretty printing need to be essentially duplicated. In $F^{\mu}_{\leq\geq}$, in addition to polymorphic constructors, we would just need to define the pretty printer for Exp_2, and that function would also work for Exp_1. Alternatively, one could define only Exp_2 and type desugar with the imprecise type $\mathsf{Exp}_2 \to \mathsf{Exp}_2$, which does not statically guarantee that the Neg constructor has been removed. This solution avoids the duplication at the cost of static typing guarantees. In $F^{\mu}_{\leq\geq}$ we do not need such compromise: we can avoid code duplication and preserve the static typing guarantees.

8.1.6 $F^{\mu}_{\leq}$: kernel $F_{\leq}$ with iso-recursive types

As Chapter 1.1.4 mentioned, no previous calculi with bounded quantification and recursive types are fully satisfactory. Equi-recursive types are quite problematic, since they can change the expressive power of the subtyping relation in unexpected ways. More importantly, adding equi-recursive subtyping to $F_{\leq}$ requires novel algorithms, and the extension is non-modular, requiring several changes to existing definitions and proofs.

Our type system directly combines kernel $F_{\leq}$ and the nominal unfolding rules together. Surprisingly, more or less, these two rules are orthogonal in our system. The addition of the nominal unfolding rules has almost no effect in the original proofs in kernel $F_{\leq}$. That is the proofs for important lemmas, such as transitivity, are nearly the same as those in kernel $F_{\leq}$, except that we need a new case to deal with recursive types. Thus proofs that have been very hard in the past, such as transitivity, are very simple in $F^{\mu}_{\leq}$.

The more challenging aspect in the metatheory of $F^{\mu}_{\leq}$ lies in the *unfolding lemma*:

$$\Gamma \vdash \mu\alpha.\, A \leq \mu\alpha.\, B \quad \Rightarrow \quad \Gamma \vdash [\alpha \mapsto \mu\alpha.\, A]\, A \leq [\alpha \mapsto \mu\alpha.\, B]\, B$$

which reveals an important property for iso-recursive types: if two iso-recursive types are subtypes, then their one-step unfoldings are also subtypes. To prove the unfolding lemma, a

generalized lemma is needed, as explained in Chapter 4.4. In $F^{\mu}_{\leq}$, we show that the previous generalized approach is insufficient, due to bounded quantification. Therefore, a more general lemma is proposed.

Another challenge is decidability. Although both kernel $F_{\leq}$ and the nominal unfolding rules (for simple calculi) have been independently proved decidable, their decidability proofs use very different measures. A natural combination is problematic, thus we need a new approach.

After overcoming those challenges, we will see that, $F^{\mu}_{\leq}$ performs well in various dimensions: it is transitive, decidable, conservative and modular. Furthermore, there is a simple, sound and complete algorithmic type system to enable implementations, and to provide important help in the proofs of results such as conservativity of typing.

Finally, we have also formalized an extension of $F^{\mu}_{\leq}$ with bottom and lower bounded quantification, called $F^{\mu}_{\leq\geq}$. All the same results that are proved for $F^{\mu}_{\leq}$ are also proved for $F^{\mu}_{\leq\geq}$, including transitivity, decidability and type soundness.

8.2 Bounded Quantification with Iso-Recursive Types

This section introduces a full calculus, called $F^{\mu}_{\leq}$, with bounded quantification, records and recursive types. $F^{\mu}_{\leq}$ is an extension of kernel $F_{\leq}$ [37] with iso-recursive types.

8.2.1 Syntax and Well-Formedness

The syntax of types and contexts for $F^{\mu}_{\leq}$ is shown below.

$$
\begin{array}{llll}
\text{Types} & A, B, \ldots & ::= & \mathsf{nat} \mid \top \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\, A \mid A^{\alpha} \mid \forall(\alpha \leq A).\, B \\
 & & & \mid \{l_i : A_i \,^{i \in 1 \cdots n}\} \\
\text{Expressions} & e & ::= & x \mid \mathsf{i} \mid e_1\, e_2 \mid \lambda x : A.\, e \mid e\, A \mid \Lambda(\alpha \leq A).\, e \\
 & & & \mid \mathsf{unfold}\, [A]\, e \mid \mathsf{fold}\, [A]\, e \mid \{l_i = e_i \,^{i \in 1 \cdots n}\} \mid e.l \\
\text{Values} & v & ::= & \mathsf{i} \mid \lambda x : A.\, e \mid \mathsf{fold}\, [A]\, v \mid \Lambda(\alpha \leq A).\, e \mid \{l_i = v_i \,^{i \in 1 \cdots n}\} \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, \alpha \leq A \mid \Gamma, x : A
\end{array}
$$

Meta-variables A, B, C, D range over types. Types consist of: natural numbers (nat), the top type ($\top$), function types ($A \to B$), type variables (α), recursive types ($\mu\alpha.\ A$), labelled types (A^{α}), universal types ($\forall(\alpha \leq A).\ B$), and record types ($\{l_i : A_i \,^{i \in 1 \cdots n}\}$). Labelled types are types that are annotated with a type variable. They are used for dealing with subtyping of iso-recursive types as part of the nominal unfolding approach. Expressions, denoted by meta-variable e, include: term variables (x), natural numbers (i), applications ($e_1\ e_2$), abstractions ($\lambda x : A.\ e$), type applications ($e\ A$), type abstractions ($\Lambda(\alpha \leq A).\ e$), fold expressions ($\mathsf{fold}\ [A]\ e$), and unfold expressions ($\mathsf{unfold}\ [A]\ e$), records ($\{l_i = e_i \,^{i \in 1 \cdots n}\}$) and record selection ($e.l$). Among them, natural numbers, abstractions and type abstractions are values. Fold expressions and records can be values if their inner expressions are also values. The context is used to store type variables with their bounds, and term variables with their types. Note that it is unnecessary to distinguish recursive variables and universal variables.

$\boxed{\Gamma \vdash A}$ *(Well-formed Type)*

$$\textsc{wft-nat}\ \frac{}{\Gamma \vdash \mathsf{nat}} \qquad \textsc{wft-Top}\ \frac{}{\Gamma \vdash \top} \qquad \textsc{wft-fvar}\ \frac{\alpha \leq A \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{wft-all}\ \frac{\Gamma \vdash A \qquad \Gamma, \alpha \leq A \vdash B}{\Gamma \vdash \forall(\alpha \leq A).\ B}$$

$$\textsc{wft-arrow}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2} \qquad \textsc{wft-rec}\ \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \mu\alpha.\ A} \qquad \textsc{wft-flabel}\ \frac{\Gamma \vdash A}{\Gamma \vdash A^{\alpha}} \qquad \textsc{wft-rcd}\ \frac{\Gamma \vdash A_i \qquad \text{for each i}}{\Gamma \vdash \{l_i : A_i\ ^{i \in 1 \cdots n}\}}$$

$\boxed{\Gamma \vdash A \leq B}$ *(Subtyping)*

$$\textsc{s-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nat} \leq \mathsf{nat}} \qquad \textsc{s-top}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash A \leq \top} \qquad \textsc{s-var}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \alpha}{\Gamma \vdash \alpha \leq \alpha} \qquad \textsc{s-arrow}\ \frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2}$$

$$\textsc{s-frec}\ \frac{\Gamma, \alpha \leq \top \vdash [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B}{\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B} \qquad \textsc{s-flabel}\ \frac{\Gamma \vdash A \leq B}{\Gamma \vdash A^{\alpha} \leq B^{\alpha}}$$

$$\textsc{s-equivall}\ \frac{\Gamma \vdash A_1 \leq A_2 \qquad \Gamma \vdash A_2 \leq A_1 \qquad \Gamma, \alpha \leq A_2 \vdash B \leq C}{\Gamma \vdash \forall(\alpha \leq A_1).\ B \leq \forall(\alpha \leq A_2).\ C} \qquad \textsc{s-vartrans}\ \frac{\Gamma \vdash B \leq A \qquad \alpha \leq B \in \Gamma}{\Gamma \vdash \alpha \leq A}$$

$$\textsc{s-rcd}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \{k_j : A_j\ ^{j \in 1 \cdots m}\} \qquad \{l_i\ ^{i \in 1 \cdots n}\} \subseteq \{k_j\ ^{j \in 1 \cdots m}\} \qquad k_j = l_i \text{ implies } \Gamma \vdash A_j \leq B_i}{\Gamma \vdash \{k_j : A_j\ ^{j \in 1 \cdots m}\} \leq \{l_i : B_i\ ^{i \in 1 \cdots n}\}}$$

Figure 8.1: Well-formedness and subtyping rules.

The definition of a well-formed environment $\vdash \Gamma$ is standard, ensuring that all variables in the environment are distinct and all types in the environment are well-formed. The top of Figure 8.1 shows the judgement for well-formed types. A type is well-formed if all of its free variables are in the context. The rules of this judgement are straightforward.

8.2.2 Subtyping

The bottom of Figure 8.1 shows the subtyping judgement. Our subtyping rules are mostly standard. The rules essentially include the rules of the algorithmic version of kernel $F_{\leq}$ [37, 35], but the rule for bounded quantification is generalized. The rules S-VAR and S-VARTRANS are standard $F_{\leq}$ rules. Note that since we do not distinguish universal and recursive variables, those rules apply also to recursive type variables. The rule for function types (rule S-ARROW) is contravariant on the input types and covariant on the output types.

The rule for bounded quantification is interesting, stating that two universal types are subtypes if their bounds are equivalent (i.e. they are subtypes of each other) and the bodies are subtypes. Note that rule S-EQUIVALL is more general than rule S-KERNELALL since the latter one requires the bounds are equal. The reason to have the more general rule using equivalent

bounds is that, for records, we wish to accept subtyping statements such as:

$$\forall(\alpha \leq \{x : \mathsf{nat}, y : \mathsf{nat}\}).\ \alpha \to \alpha \leq \forall(\alpha \leq \{y : \mathsf{nat}, x : \mathsf{nat}\}).\ \alpha \to \alpha$$

where the bounds can be syntactically different, but equivalent types. In the presence of records or other features (such as intersection and union types [104, 51, 15]) we can have such equivalent, but not syntactically equal types. Therefore, we should generalize the rule for bounded quantification do deal with those cases. This generalization to equivalent bounds retains decidable subtyping just as kernel $F_{\leq}$ as we shall see in Chapter 8.3.2.

For dealing with iso-recursive subtyping we employ the nominal unfolding rules (rule S-FNOMINAL), which have equivalent expressive power to the iso-recursive Amber rules. The nominal unfolding rules have been discussed in Chapter 4.5. The reason to choose the nominal unfolding rules is that they enable us to prove important metatheoretical results easily, such as transitivity and develop an algorithmic formulation of subtyping.

We extend the rule S-FNOMINAL to the rule S-FREC in $F^{\mu}_{\leq}$, by bounding recursive variables with $\top$ when they are introduced into the context.

Therefore, recursive variables are also treated as universal variables, and we do not need to adjust the form of contexts in $F_{\leq}$ for $F^{\mu}_{\leq}$. Apart from this, no other changes are necessary, making the addition of recursive types mostly non-invasive. Consequently, the proofs of narrowing, reflexivity and transitivity are the same as the original one for $F_{\leq}$, except for the new cases dealing with recursive types and minor adjustments to the rule of bounded quantification due to the generalization to equivalent bounds.

Lemma 95 (Narrowing). If $\Gamma_1 \vdash C \leq C'$ and $\Gamma_1, \alpha \leq C', \Gamma_2 \vdash A \leq B$ then $\Gamma_1, \alpha \leq C, \Gamma_2 \vdash A \leq B$.

Theorem 96 (Reflexivity). If $\vdash \Gamma$ and $\Gamma \vdash A$ then $\Gamma \vdash A \leq A$.

Theorem 97 (Transitivity). If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$.

Another important lemma is the *unfolding lemma*, which reveals that: if two recursive types are subtypes, then their unfoldings are also subtypes. The unfolding lemma is important for proving preservation in a system with iso-recursive subtyping. A key difficulty in the formalization of $F^{\mu}_{\leq}$ is proving the unfolding lemma which, due to the presence of bounded quantification, requires a different proof technique compared to the proofs in Chapter 4. We discuss the proof of the unfolding lemma in Chapter 8.3.1.

Lemma 98 (Unfolding Lemma). If $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B$ then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B$.

8.2.3 Type Soundness

Figure 8.2 and 8.3 show the typing rules and reduction rules. Most rules are standard except for the typing rule for unfold and fold. For these two expressions we use structural rules instead (rule TYPING-SUNFOLD and rule TYPING-SFOLD), as we explained in Chapter 8.1.6.

$\boxed{\Gamma \vdash e : A}$ *(Typing)*

$$\textsc{typing-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{i} : \mathsf{nat}} \qquad \textsc{typing-var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{typing-sub}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}$$

$$\textsc{typing-abs}\ \frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash \lambda x : A_1.\ e : A_1 \to A_2} \qquad \textsc{typing-sfold}\ \frac{\Gamma \vdash e : [\alpha \mapsto B]\ A \qquad \Gamma \vdash \mu\alpha.\ A \leq B}{\Gamma \vdash \mathsf{fold}\ [B]\ e : B}$$

$$\textsc{typing-sunfold}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq \mu\alpha.\ B}{\Gamma \vdash \mathsf{unfold}\ [A]\ e : [\alpha \mapsto A]\ B} \qquad \textsc{typing-app}\ \frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2}$$

$$\textsc{typing-proj}\ \frac{\Gamma \vdash e : \{l_i : A_i\ ^{i \in 1 \cdots n}\}}{\Gamma \vdash e.l_i : A_i} \qquad \textsc{typing-tabs}\ \frac{\Gamma, \alpha \leq A \vdash e : B}{\Gamma \vdash \Lambda(\alpha \leq A).\ e : \forall(\alpha \leq A).\ B}$$

$$\textsc{typing-tapp}\ \frac{\Gamma \vdash e : \forall(\alpha \leq B_1).\ B_2 \qquad \Gamma \vdash A \leq B_1}{\Gamma \vdash e\ A : [\alpha \mapsto A]\ B_2} \qquad \textsc{typing-rcd}\ \frac{\text{for each } i \quad \Gamma \vdash e_i : A_i}{\Gamma \vdash \{l_i = e_i\ ^{i \in 1 \cdots n}\} : \{l_i : A_i\ ^{i \in 1 \cdots n}\}}$$

Figure 8.2: Typing Rules.

Structural unfolding lemma Since the typing rules that we adopt for fold/unfold expressions are the structural rules, which generalize the conventional rules, we need a more general form for the unfolding lemma. The generalization of the lemma is necessary for the type preservation proof with the structural folding/unfolding rules. We call the new lemma the *structural unfolding lemma*:

Lemma 99 (Structural unfolding lemma)**.** If $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ C \leq \mu\alpha.\ D \leq \mu\alpha.\ B$ then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$.

Proof. By applying transitivity (Theorem 97) to the 3 conditions below:

1. $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ C$ implies $\Gamma, \alpha \leq \top \vdash A \leq C$ by Lemma 107, implies $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ C]\ C$ by substitution lemma.
2. $\Gamma \vdash \mu\alpha.\ C \leq \mu\alpha.\ D$ implies $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ C]\ C \leq [\alpha \mapsto \mu\alpha.\ D]\ D$ by unfolding lemma (Lemma 98).
3. $\Gamma \vdash \mu\alpha.\ D \leq \mu\alpha.\ B$ implies $\Gamma, \alpha \leq \top \vdash D \leq B$ by Lemma 107, implies $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ D]\ D \leq [\alpha \mapsto \mu\alpha.\ D]\ B$ by substitution lemma.

□

In this lemma, in the one-step unfolding the recursive types substituted in the bodies are, respectively, a supertype and a subtype of $\mu\alpha.\ A$ and $\mu\alpha.\ B$. In contrast, in the unfolding lemma in previous chapters (Lemma 8, 26, 81, etc.), the recursive types that get substituted in

$\boxed{e_1 \hookrightarrow e_2}$ *(Reduction)*

$$\textsc{step-beta}\ \frac{}{(\lambda x : A.\ e_1)\ v_2 \hookrightarrow [x \mapsto v_2]\ e_1} \qquad \textsc{step-appl}\ \frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2} \qquad \textsc{step-appr}\ \frac{e_2 \hookrightarrow e_2'}{v_1\ e_2 \hookrightarrow v_1\ e_2'}$$

$$\textsc{step-fld}\ \frac{}{\mathsf{unfold}\ [A]\ (\mathsf{fold}\ [B]\ v) \hookrightarrow v} \qquad \textsc{step-unfold}\ \frac{e \hookrightarrow e'}{\mathsf{unfold}\ [A]\ e \hookrightarrow \mathsf{unfold}\ [A]\ e'} \qquad \textsc{step-fold}\ \frac{e \hookrightarrow e'}{\mathsf{fold}\ [A]\ e \hookrightarrow \mathsf{fold}\ [A]\ e'}$$

$$\textsc{step-tapp}\ \frac{e_1 \hookrightarrow e_2}{e_1\ A \hookrightarrow e_2\ A} \qquad \textsc{step-tabs}\ \frac{}{(\Lambda(\alpha \leq A).\ e)\ B \hookrightarrow [\alpha \mapsto B]\ e} \qquad \textsc{step-proj}\ \frac{e \hookrightarrow e'}{e.l_j \hookrightarrow e'.l_j}$$

$$\textsc{step-projrcd}\ \frac{}{\{l_i = v_i\ ^{i \in 1 \cdots n}\}.l_j \hookrightarrow v_j}$$

$$\textsc{step-rcd}\ \frac{e_j \hookrightarrow e_j'}{\{l_i = v_i\ ^{i \in 1 \cdots j-1},\ l_j = e_j,\ l_k = e_k\ ^{k \in j+1 \cdots n}\} \hookrightarrow \{l_i = v_i\ ^{i \in 1 \cdots j-1},\ l_j = e_j',\ l_k = e_k\ ^{k \in j+1 \cdots n}\}}$$

Figure 8.3: Reduction Rules.

the bodies are the same. As Chapter 8.3.1 will discuss, both forms of the unfolding lemma can be proved using a more general lemma.

Type Soundness To see how the structural unfolding lemma is used in the proof of type preservation, assume that C' is $\mu\alpha.\ C$ and D' is $\mu\alpha.\ D$, let us consider such expression $\mathsf{unfold}[D'](\mathsf{fold}[C']\ e)$.

$$\textsc{typing-sunfold}\ \frac{\textsc{typing-sub}\ \dfrac{\textsc{typing-sfold}\ \dfrac{\Gamma \vdash e : [\alpha \mapsto C']\ A \qquad \Gamma \vdash \mu\alpha.\ A \leq C'}{\Gamma \vdash \mathsf{fold}[C]\ e : C'} \qquad \Gamma \vdash C' \leq D'}{\Gamma \vdash \mathsf{fold}[C]\ e : D'} \qquad \Gamma \vdash D' \leq \mu\alpha.\ B}{\Gamma \vdash \mathsf{unfold}[D'](\mathsf{fold}[C']\ e) : [\alpha \mapsto D']\ B}$$

Starting from a closed expression, both C' and D' must be recursive types. The type of $\mathsf{unfold}[D'](\mathsf{fold}[C']\ e)$ becomes $[\alpha \mapsto \mu\alpha.\ D]\ B$, and it should be a subtype of $[\alpha \mapsto \mu\alpha.\ C]\ A$, which is the type of reduction result e.

The other parts of the type soundness proof are standard, thus we have:

Theorem 100 (Preservation). If $\vdash e : A$ and $e \hookrightarrow e'$ then $\vdash e' : A$.

Theorem 101 (Progress). If $\vdash e : A$ then e is a value or exists $e', e \hookrightarrow e'$.

8.2.4 Algorithmic typing

The rules that we have presented in Figure 8.2 are declarative. The subsumption rule overlaps in type syntax with all other rules, making it non-trivial to derive an implementation from the rules.

$\boxed{\Gamma \vdash A \Uparrow B}$ *(Upper Exposure)*

$$\textsc{XA-promote}\ \frac{\alpha \leq A \in \Gamma \qquad \Gamma \vdash A \Uparrow B}{\Gamma \vdash \alpha \Uparrow B} \qquad \textsc{XA-up}\ \frac{A \text{ is not type variable}}{\Gamma \vdash A \Uparrow A}$$

$\boxed{\Gamma \vdash A \Downarrow B}$ *(Lower Exposure)*

$$\textsc{XA-top}\ \frac{}{\Gamma \vdash \top \Downarrow \mu\alpha.\, \top} \qquad \textsc{XA-down}\ \frac{A \text{ is not type variable or } \top}{\Gamma \vdash A \Downarrow A}$$

$\boxed{\Gamma \vdash_a e : A}$ *(Algorithmic Typing)*

$$\textsc{atyp-app}\ \frac{\Gamma \vdash_a e_1 : A \qquad \Gamma \vdash A \Uparrow A_1 \to A_2 \qquad \Gamma \vdash_a e_2 : B \qquad \Gamma \vdash B \leq A_1}{\Gamma \vdash_a e_1\ e_2 : A_2}$$

$$\textsc{atyp-tapp}\ \frac{\Gamma \vdash_a e : B \qquad \Gamma \vdash B \Uparrow \forall(\alpha \leq B_1).\, B_2 \qquad \Gamma \vdash A \leq B_1}{\Gamma \vdash_a e\, A : [\alpha \mapsto A]\, B_2}$$

$$\textsc{atyp-sunfold}\ \frac{\Gamma \vdash_a e : A \qquad \Gamma \vdash B \Uparrow \mu\alpha.\, C \qquad \Gamma \vdash A \leq B}{\Gamma \vdash_a \mathsf{unfold}\ [B]\ e : [\alpha \mapsto B]\, C}$$

$$\textsc{atyp-sfold}\ \frac{\Gamma \vdash_a e : A \qquad \Gamma \vdash C \Downarrow \mu\alpha.\, B \qquad \Gamma \vdash A \leq [\alpha \mapsto C]\, B}{\Gamma \vdash_a \mathsf{fold}\ [C]\ e : C}$$

Figure 8.4: Algorithmic Typing.

Figure 8.4 shows the algorithmic rules for typing. We only present new rules and rules that differ from Figure 8.2. Compared with the declarative typing rules, the subsumption rule (TYPING-SUB) is removed. Besides, application (TYPING-APP), type application (TYPING-TAPP), structural folding (TYPING-SFOLD) and structural unfolding (TYPING-SUNFOLD) rules are replaced by rules ATYP-APP, ATYP-TAPP, ATYP-SFOLD and ATYP-SUNFOLD, respectively. In the algorithmic typing rules we take the standard approach of distributing subtyping checks in appropriate places in the other rules, thus eliminating the need for the subsumption rule.

One interesting point is the two exposure relations $\Uparrow$ and $\Downarrow$ in $F^{\mu}_{\leq}$. In $F_{\leq}$, there is only the *upper exposure* function ($\Gamma \vdash A \Uparrow B$), which is used to find the least non-variable upper bound for a variable in the context [100]. Thus, the upper exposure function plays an important role for finding the minimal type with the algorithmic typing rules. To make our rules more general, we additionally define the *lower exposure* function ($\Gamma \vdash A \Downarrow B$) to find the greatest non-variable subtype B for A. For $F^{\mu}_{\leq}$, lower exposure only helps to find the correct shape for the recursive type body to be folded in rule ATYP-SFOLD by mapping $\top$ to $\mu\alpha.\, \top$. The lower exposure function will be more useful when we have lower bounded variables in the system, as we will see in Chapter 8.4.

The algorithmic rules are equivalent (sound and complete) with respect to the declarative rules:

Theorem 102 (Soundness of the algorithmic rules)**.** If $\Gamma \vdash_a e : A$ then $\Gamma \vdash e : A$.

Theorem 103 (Completeness of the algorithmic rules)**.** If $\Gamma \vdash e : A$ then there exists a B such that $\Gamma \vdash_a e : B$ and $\Gamma \vdash B \leq A$.

Theorem 103 implies that our algorithm can always find a minimal type, which is an important property for $F^{\mu}_{\leq}$.

8.3 Metatheory of $F^{\mu}_{\leq}$

In this section we discuss the most interesting and difficult aspects of the metatheory of $F^{\mu}_{\leq}$ in more detail. We cover three key properties: the *unfolding lemma*, *decidability* of subtyping and the *conservativity* of $F^{\mu}_{\leq}$ over the original $F_{\leq}$. The interaction between iso-recursive types and bounded quantification requires significant changes in the proofs of the unfolding lemma and decidability. In addition, conservativity cannot be proved using a declarative formulation of $F^{\mu}_{\leq}$, and we need to employ the algorithmic formulation instead.

8.3.1 Unfolding lemma

The unfolding lemma (Lemma 98) is a core lemma for the metatheory of a calculus with iso-recursive subtyping. Though the statement of the unfolding lemma is quite simple and intuitive to understand, the lemma cannot be proved directly. We will firstly review the previous approach to prove the unfolding lemma, which does not account for bounded quantification, and then show how to transfer this approach to a system with bounded quantification.

The previous approach for proving the unfolding lemma. Because the premise of the unfolding lemma is a subtyping relation between two recursive types $\mu\alpha.\ A \leq \mu\alpha.\ B$, a direct induction on such subtyping relation is problematic. Thus, we need to prove a generalized lemma first. Following the idea from Chapter 4 that lemma would have the following form:

Lemma 104 (The generalized unfolding lemma for nominal unfoldings in Chapter 4)**.** If we have $\Gamma_1, \alpha, \Gamma_2, \vdash A \leq B$ and $\Gamma_1 \vdash \mu\alpha.\ C \leq \mu\alpha.\ D$ then

1. $\Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$ implies $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$;
2. $\Gamma_1, \alpha, \Gamma_2 \vdash [\alpha \mapsto D^{\alpha}]\ A \leq [\alpha \mapsto C^{\alpha}]\ B$ implies $\Gamma_1, \Gamma_2 \vdash [\alpha \mapsto \mu\alpha.\ D]\ A \leq [\alpha \mapsto \mu\alpha.\ C]\ B$.

Due to the tricky interaction between rule S-VAR and rule S-ARROW, in the generalized unfolding lemma we need two mutual dependent lemmas: one is used for covariant cases (1) and the other one is used for contravariant cases (2). The proof for this lemma proceeds by induction on $\Gamma_1, \alpha, \Gamma_2, \vdash A \leq B$. In the inductive proof we need to switch between covariance and contravariance. In particular, in the rule S-ARROW case for functions, we need an induction

hypothesis that arises from conclusion (2) to prove the contravariant premise $\Gamma \vdash B_1 \leq A_1$ relating the input types of the function.

For the generalized unfolding lemma in $F^{\mu}_{\leq}$, Lemma 104 is unfortunately not general enough. In a setting with bounded quantification, Γ_2 may contain bounds with the type variable α, and those variables are not being substituted in Lemma 104. Let us consider the effect of adding rule S-VARTRANS. In the conclusion of Lemma 104, the variable α is substituted by another type and tracked in the context Γ_2. In the premises of rule S-VARTRANS, we need to find the upper bound of variable α in the contexts Γ_1 and Γ_2. With those two observations, in our new attempt, the context also needs to do substitutions. Thus, a natural attempt to solve this problem is to reformulate the lemma into the following form (for the covariant case (1)):

Proposition 105 (A first attempt at the generalized unfolding lemma)**.** If $\Gamma_1, \alpha \leq \top, \Gamma_2, \vdash A \leq B$ and $\Gamma_1 \vdash \mu\alpha.\, C \leq \mu\alpha.\, D$ then $\Gamma_1, \alpha \leq \top, \Gamma_2[\alpha \mapsto ?^{\alpha}] \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$ implies $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\, ?] \vdash [\alpha \mapsto \mu\alpha.\, C]\ A \leq [\alpha \mapsto \mu\alpha.\, D]\ B$.

Here, the syntax $\Gamma[\alpha \mapsto S]$ denotes that all the occurrences of α in context Γ will be replaced by a specified type S. However, we do not know yet what type should be filled in the hole ? in Proposition 105, so we leave the hole there for now. Although we omit conclusion (2) in Proposition 105, similar reasoning applies to that conclusion.

Let us now consider the effect of adding the rule S-EQUIVALL: Assume that $A := \forall(\beta \leq U_1).\, T_1$ and $B := \forall(\beta \leq U_2).\, T_2$. The goal would look like:

$$\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\, ?] \vdash [\alpha \mapsto \mu\alpha.\, C]\ \forall(\beta \leq U_1).\, T_1 \leq [\alpha \mapsto \mu\alpha.\, D]\ \forall(\beta \leq U_2).\, T_2$$

After simplification and applying rule S-EQUIVALL, one of the goals becomes:

$$\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\, ?], \beta \leq [\alpha \mapsto \mu\alpha.\, D]\ U_2 \vdash [\alpha \mapsto \mu\alpha.\, C]\ T_1 \leq [\alpha \mapsto \mu\alpha.\, D]\ T_2$$

From the above, we would expect that the hole ? is filled with D because all the substitutions in the context must be the same in order to apply induction hypothesis. Thus, a second attempt at the generalized unfolding lemma looks like:

Proposition 106 (The second attempt at the generalized unfolding lemma (1))**.** If $\Gamma_1, \alpha \leq \top, \Gamma_2, \vdash A \leq B$ and $\Gamma_1 \vdash \mu\alpha.\, C \leq \mu\alpha.\, D$ then $\Gamma_1, \alpha \leq \top, \Gamma_2[\alpha \mapsto D^{\alpha}] \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$ implies $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\, D] \vdash [\alpha \mapsto \mu\alpha.\, C]\ A \leq [\alpha \mapsto \mu\alpha.\, D]\ B$.

Proposition 106 deals with rule S-VARTRANS and S-EQUIVALL successfully. However, the function case, which is correctly proven in Lemma 104, will break. Consider $A := A_1 \to A_2$ and $B := B_1 \to B_2$, and apply rule S-ARROW. We need to prove two subgoals:

1. $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\, D] \vdash [\alpha \mapsto \mu\alpha.\, C]\ A_2 \leq [\alpha \mapsto \mu\alpha.\, D]\ B_2$;
2. $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\, D] \vdash [\alpha \mapsto \mu\alpha.\, D]\ B_1 \leq [\alpha \mapsto \mu\alpha.\, C]\ A_1$.

Note that we do not have any induction hypothesis for proving subgoal (2) because occurrences of α in Γ_2 have been substituted by $\mu\alpha.\, D$, but we expect the α's to have been replaced by $\mu\alpha.\, C$ for applying the induction hypothesis. Even if we add the second conclusion back to the

Proposition 106, we still have problems. For conclusion (2), the type used for the substitution in the context should be same as the type used for the substitution in the right-hand side of the subtyping. If we fill the hole ? with C in Proposition 105, the subgoal (1) in case S-ARROW will get stuck for a similar reason. Therefore, the type in the hole ? cannot be the type C or D.

In summary, in the previous approach showed in Chapter 4, without bounded quantification, only the interaction of covariance/contravariance between types has to be considered. In contrast, with bounded quantification, the interaction of covariance/contravariance among contexts and types also needs to be considered. Our generalization should be able to deal with all the complications arising from rule S-VAR, rule S-VARTRANS, rule S-ARROW and rule S-EQUIVALL.

The generalized unfolding lemma for $F^{\mu}_{\leq}$. Surprisingly, the generalization is relatively straightforward. For solving the issue we mention above, instead of picking type C or type D, we pick an intermediate type S between C and D. We need the following auxiliary lemma:

Lemma 107. If $\Gamma, \alpha \leq \top \vdash [\alpha \mapsto A^{\alpha}]\ A \leq [\alpha \mapsto B^{\alpha}]\ B$ then $\Gamma, \alpha \leq \top \vdash A \leq B$.

The generalized unfolding lemma for $F^{\mu}_{\leq}$ is:

Lemma 108 (The generalized unfolding lemma for $F^{\mu}_{\leq}$)**.** If (1) $\Gamma_1, \alpha \leq \top,\ \Gamma_2 \vdash A \leq B$, (2) $\Gamma_1 \vdash \mu\alpha.\ C \leq \mu\alpha.\ S$ and (3) $\Gamma_1 \vdash \mu\alpha.\ S \leq \mu\alpha.\ D$ then

1. $\Gamma_1, \alpha \leq \top,\ \Gamma_2[\alpha \mapsto S^{\alpha}] \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$ implies $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\ S] \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$;
2. $\Gamma_1, \alpha \leq \top,\ \Gamma_2[\alpha \mapsto S^{\alpha}] \vdash [\alpha \mapsto D^{\alpha}]\ A \leq [\alpha \mapsto C^{\alpha}]\ B$ implies $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\ S] \vdash [\alpha \mapsto \mu\alpha.\ D]\ A \leq [\alpha \mapsto \mu\alpha.\ C]\ B$.

By applying Lemma 108 with $\Gamma_1 = \Gamma$, $\Gamma_2 = \cdot$, $S = A$, $C = A$ and $D = B$, we prove the unfolding lemma (Lemma 98). The premises can be obtained by inversion on $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B$, and then applying Lemma 107. Next we give an overview of the key points in the proof of Lemma 108 for the four tricky cases. Assume that we do induction on premise (1).

- Rule S-VAR: This case mostly follows the previous approach. The interesting point is that we cannot obtain $\mu\alpha.\ C \leq \mu\alpha.\ D$ directly from the conditions raised by the goals, but have one extra step via transitivity (Theorem 97).
- Rule S-VARTRANS: For this case we need to consider two subcases. If $A = \alpha$ then from the context $\alpha \leq \top$ we can easily know that B is $\top$ too. Otherwise, we apply the induction hypothesis directly.
- Rule S-ARROW: This case is surprisingly easy: we can follow the previous approach for dealing with the contravariant case for subtyping input types of functions, and apply the induction hypothesis derived from conclusion (2) directly. Note that the key point of avoiding the issue we discussed before is that, by picking an intermediate type S, we decouple the substitution in the context and in the subtyping relation for the function case. That is, the substitution with type $\mu\alpha.\ S$ is invariant in both subgoals, independent of the substitution in the subtyping relation.

- Rule S-EQUIVALL: This case is the most interesting one. Assume $A = \forall(\beta \leq U_1).\ T_1$ and $B = \forall(\beta \leq U_2).\ T_2$. Let us consider how to prove the goal (1). Doing inversion on condition arising from the goal, we obtain that $[\alpha \mapsto C^{\alpha}]\ U_1$ and $[\alpha \mapsto D^{\alpha}]\ U_2$ are equivalent. Meanwhile, we know U_1 and U_2 that are equivalent. There are two possibilities: either C and D are equivalent or α is not in type U_1 nor U_2. For the latter case, we can rewrite the substituted types in the contexts for aligning the contexts, then solve the issue we discussed above. The former case is quite subtle: since S lies in the middle of C and D, the three types are all equivalent. In other words, we can change the substituted types arbitrarily and get the equivalent types and contexts. The critical point is that, although the substitution in the context is indeed affected by the substitution in the supertype, since the bounds are equivalent, the type S will converge into the types C and D.

Remark Some readers might want to know if we can combine the bounded quantification with double unfolding rule. Unfortunately, the unfolding lemma cannot be proved successfully. Our generalized unfolding lemma relies on the inversion lemma for rule S-VARTRANS:

Lemma 109. If $\alpha \leq A \in \Gamma, \Gamma \vdash \alpha \leq B$, and $\alpha \neq B$ then $\Gamma \vdash A \leq B$.

but this inversion lemma cannot be correctly applied when double unfolding rule is used.

Assume we have a generalized unfolding lemma (as Lemma 108) for double unfoldings. By omitting the second part, it is like:

Proposition 110 (The generalized unfolding lemma with double unfoldings)**.** If

1. $\Gamma_1, \alpha \leq \top,\ \Gamma_2 \vdash A \leq B$;
2. $\Gamma_1 \vdash \mu\alpha.\ C \leq \mu\alpha.\ S$ and $\Gamma_1 \vdash \mu\alpha.\ S \leq \mu\alpha.\ D$;
3. $\Gamma_1, \alpha \leq \top,\ \Gamma_2[\alpha \mapsto S] \vdash [\alpha \mapsto C]\ A \leq [\alpha \mapsto D]\ B$

then $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\ S] \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$.

Assume we do induction on premise (1) and deal with the case rule S-VARTRANS. What if when $A := \beta$, $B := \alpha$, $\beta \leq U \in \Gamma_2$ and $\alpha \neq \beta$? Assume premise (2) holds and $D := \beta$. Then, the goal becomes $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\ S] \vdash \beta \leq [\alpha \mapsto \mu\alpha.\ D]\ B$. We might want to apply rule S-VARTRANS to the goal, and then we need to prove $\Gamma_1, \Gamma_2[\alpha \mapsto \mu\alpha.\ S] \vdash [\alpha \mapsto \mu\alpha.\ S]\ U \leq [\alpha \mapsto \mu\alpha.\ D]\ B$. By applying induction hypothesis, we need to prove $\Gamma_1, \Gamma_2[\alpha \mapsto S] \vdash [\alpha \mapsto S]\ U \leq [\alpha \mapsto D]\ B$, which is raised from premise (3). Premise (3) now is $\Gamma_1, \alpha \leq \top,\ \Gamma_2 \vdash \beta \leq [\alpha \mapsto D]\ B$, and we want to prove the new goal by applying inversion lemma (Lemma 109) on it. However, since $B := \alpha$ and $D := \beta$, premise (3) becomes $\Gamma_1, \alpha \leq \top,\ \Gamma_2 \vdash \beta \leq \beta$, thus we cannot apply inversion lemma (Lemma 109) to it.

Note that although we get stuck in Proposition 110, it does not necessarily mean the unfolding lemma is not true. We have no idea how to find an alternative generalized unfolding lemma for double unfoldings, thus we leave this as the future work.

8.3.2 Decidability

Decidability is one of the important properties of $F^{\mu}_{\leq}$. We first start by reviewing the approaches to prove decidability in kernel $F_{\leq}$, and nominal unfoldings, and then describe our approach to prove decidability. These two previous approaches to prove decidability employ different measures, which creates a challenge for proving the decidability of $F^{\mu}_{\leq}$.

Decidability of kernel $F_{\leq}$ It is well-known that bounded quantification for full $F_{\leq}$ is undecidable [98]. However, for the kernel $F_{\leq}$, identical bounds make the system decidable. A common practice is to define a *weight* function to compute the size of a type [100]. The interesting cases are for type variables, universal and function types:

$$\begin{array}{lcl} \mathit{weight}_{\Gamma}(\top) & = & 1 \\ \mathit{weight}_{\Gamma_1,\alpha\leq A,\Gamma_2}(\alpha) & = & 1+\mathit{weight}_{\Gamma_1}(A) \\ \mathit{weight}_{\Gamma}(\forall(\alpha\leq A).\,B) & = & 1+\mathit{weight}_{\Gamma,\alpha\leq A}(B) \\ \mathit{weight}_{\Gamma}(A\to B) & = & 1+\mathit{weight}_{\Gamma}(A)+\mathit{weight}_{\Gamma}(B) \end{array}$$

For universal types, we store its bound into a context Γ, and when we meet the universal variable, we retrieve its bound from the context and compute the size recursively. Since the size of a conclusion is always greater than any premise, this measure can be used to show that the subtyping algorithm in kernel $F_{\leq}$ terminates for all inputs.

Decidability of nominal unfoldings. The nominal unfolding rule in simple calculi with subtyping is also decidable, as Chapter 4.6 showed. Compared with kernel $F_{\leq}$, the decidability proof of nominal unfoldings is trickier. In Chapter 4.6, we choose a size measure based on an over-approximation of the height of the fully unfolded tree. We repeat it again for convenience. In brief, we define a *height* function, whose variable, function and universal cases are:

$$\begin{array}{lcl} \mathit{height}_{\Psi}(\top) & = & 0 \\ \mathit{height}_{\Psi}(\alpha) & = & \Psi(\alpha) \text{ if } \alpha\in\Psi \text{ else } 0 \\ \mathit{height}_{\Psi}(A\to B) & = & 1+\max(\mathit{height}_{\Psi}(A),\mathit{height}_{\Psi}(B)) \\ \mathit{height}_{\Psi}(\mu\alpha.\,A) & = & 1+\text{let } i=\mathit{height}_{\Psi,\alpha\mapsto 0}(A) \text{ in } \mathit{height}_{\Psi,\alpha\mapsto i+1}(A) \end{array}$$

The size measure of a type A is defined as $\mathit{height}(A)$ where the context is empty. In contrast to kernel $F_{\leq}$, the context here is used to store the size of the corresponding recursive variables. We proved that such *height* measure will precisely decrease by one for every nominal unfolding.

Decidability of $F^{\mu}_{\leq}$ Now consider how to combine these two approaches together. We wish to extend the measure of nominal unfoldings with the measure of kernel $F_{\leq}$ non-invasively. The easiest thing to do is to switch the maximum function to addition for the function case in the measure of nominal unfoldings, which simply widens the over-approximation. Then we consider the major differences between the two approaches. There are three main challenges:

- **The measures for variables are inconsistent in the two approaches**: In the *height* function, the type variable case is a base case, while in the *weight* function we will continue the

computation for a variable by getting its bound from the contexts. In $F^{\mu}_{\leq}$, a recursive variable is regarded as a universal variable, so the new formulation should reflect those two distinct situations.

- **The purposes of contexts are inconsistent in two approaches**: The context of the *height* function is used to store the pre-computed size of the recursive type body so that measures of recursive variables are counted in their nominal unfolded form. The context of the *weight* function is a straightforward bookkeeping of universal bounds. In later computation, these bounds will be retrieved and their measure will be computed to serve as the measure of a universal variable. This is to ensure that the premises in rule S-VARTRANS have a smaller measure of types than the conclusion does. We need to treat both designs carefully if we want a unified context.
- **The measure information is lost in the bounded quantification case**: Recall that we employ the rule S-EQUIVALL instead of the standard rule S-KERNELALL for $F_{\leq}$. Since we impose equivalent bounds for kernel $F_{\leq}$, for the subtyping relation $\Gamma \vdash \forall(\alpha \leq A_1).\ B_1 \leq \forall(\alpha \leq A_2).\ B_2$, the measure would consist of the measures of type A_1, A_2, B_1 and B_2. However, the measure for the premise $\Gamma, \alpha \leq A_2 \vdash B_1 \leq B_2$ will lose the measure of type A_1 because we do not store it.

We first show the measure used for the decidability of $F^{\mu}_{\leq}$, and then discuss how it addresses the concerns above. The measure is relatively simple and based on the approach from Chapter 4.6. Similarly, we define a context $\Psi := \cdot \mid \alpha \mapsto i$ which is used to store the measures of (both universal and recursive) variables during the measure computation, where i represents a natural number. Then, a measure function $size_{\Psi}(A)$, defined on types, is:

$$
\begin{array}{lcl}
size_{\Psi}(\mathsf{nat}) & = & 1 \\
size_{\Psi}(\top) & = & 1 \\
size_{\Psi}(A \to B) & = & 1 + size_{\Psi}(A) + size_{\Psi}(B) \\
size_{\Psi}(A^{\alpha}) & = & 1 + size_{\Psi}(A) \\
size_{\Psi}(\alpha) & = & 1 + \begin{cases} \Psi(\alpha) & \alpha \in \Psi \\ 0 & \alpha \notin \Psi \end{cases} \\
size_{\Psi}(\forall(\alpha \leq A).\ B) & = & \text{let } i := size_{\Psi}(A) \text{ in } 1 + i + size_{\Psi, \alpha \mapsto i}(B) \\
size_{\Psi}(\mu\alpha.\ A) & = & \text{let } i := size_{\Psi, \alpha \mapsto 1}(A) \text{ in } 1 + size_{\Psi, \alpha \mapsto i}(A)
\end{array}
$$

The formulation of the *size* function is very similar to the *height* function. We have an extra rule for universal types, and slightly adjust the variable and recursive cases. The measure of universal types is the sum of the measure of the bound and the measure of the body. For variables, one is added when they are retrieved. Accordingly, we do not need to add one when storing the *size* of recursive variables into the context. For atomic constructs, we follow the *weight* function and measure them as 1.

We solve the first challenge in a straightforward way: there is no need to distinguish between recursive and universal variables. The fact that all recursive variables in the context are bounded by a top type whose measure is simply one fits our needs naturally.

As for the second concern, despite the different purposes of contexts, the key ideas of measuring types in kernel $F_{\leq}$ and nominal unfoldings are the same: they both relate the measure of a variable to its true meaning, either its unfolded form or its bound size. A slight modification is made based on the definition of *weight*. In the *weight* function, for a universal variable, its bound is first retrieved and then the measure is computed. To align with the "pre-computation" mechanism of measuring nominal unfoldings ($i := size_{\Psi,\alpha\mapsto 1}(A)$), we also pre-compute the measure of the bound ($i := size_{\Psi}(A)$) in the *size* function, so that we retrieve the measure instead of the type bound from the context. In a well-formed type, variables are guaranteed to be unique, so we can use a single context Ψ to store the measures for both recursive variables and universal variables.

A subtler issue lies for variables in the initial subtyping context. When measuring nominal unfoldings, the context in a subtyping relation is simply a list of variables, without any bound information, so variables that occur freely can be counted as 0. In contrast, now the subtyping context stores the bound information, and the measures of bounds play a role in deciding the subtyping relation. To address this issue, we need to make sure that the bound information is pre-computed in the measure function. We transform a subtyping context into an environment containing measures Ψ, which tracks universal variables. In our decidability proof statement (Lemma 111), Ψ is computed from the subtyping context Γ by an evaluation function $eval : \Gamma \hookrightarrow \Psi$, defined as:

$$\begin{array}{lcl} eval(\cdot) & = & \cdot \\ eval(\Gamma', x : A) & = & eval(\Gamma') \\ eval(\Gamma', \alpha \leq A) & = & \text{let } \Psi' = eval(\Gamma') \text{ in } \Psi', \alpha \mapsto size_{\Psi'}(A) \end{array}$$

With both *eval* and *size* we can then state the decidability theorem:

Lemma 111. If $size_{eval(\Gamma)}(A) + size_{eval(\Gamma)}(B) \leq k$ then $\Gamma \vdash A \leq B$ is either true or not.

Theorem 112 (Decidability)**.** $\Gamma \vdash A \leq B$ is decidable.

As for the third concern, note that in $F_{\leq}$, the subtyping relation is antisymmetric [13]. Adding recursive types does not change the property of antisymmetry. However, the addition of records makes the subtyping relation not antisymmetric: two record types may be syntactically different. The lack of antisymmetry poses a challenge for our decidability proof, in particular for rule S-ALLEQUIV. However, the fields of two equivalent records must be a permutation of each other. Therefore, the measures of two equivalent record types are the same. As a result, the measure of two equivalent bounds A_1 and A_2 is equal, as Lemma 113 describes. The measure information of type A_1 can therefore be reconstructed from type A_2, addressing the final concern with decidability.

Lemma 113. If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq A$ then $size_{eval(\Gamma)}(A) = size_{eval(\Gamma)}(B)$.

8.3.3 Conservativity

One important feature of $F^{\mu}_{\leq}$ is that it is conservative over kernel $F_{\leq}$. Conservativity means that equivalent $F_{\leq}$ judgements in $F^{\mu}_{\leq}$ should behave in the same way as in $F_{\leq}$. For instance, if a subtyping statement is valid in $F_{\leq}$, then it should also be valid in $F^{\mu}_{\leq}$. Dually, if a subtyping statement over $F_{\leq}$-types is invalid in $F_{\leq}$, then it should also be invalid in $F^{\mu}_{\leq}$. In some calculi, including extensions of $F_{\leq}$ with *equi*-recursive types [65], conservativity is lost after the addition of new features.

For avoiding ambiguity, let $\vdash_F \Gamma$, $\Gamma \vdash_F A$, $\Gamma \vdash_F A \leq B$, $\vdash_F e$ and $\Gamma \vdash_F e : A$ represent the well-formedness of environment, well-formedness of types, subtyping, well-formedness of expressions and the typing relation, respectively, in kernel $F_{\leq}$. Note that all these judgements are essentially subsets of the judgements introduced in Chapter 8.2, except that the rules involving records and recursive types are removed, and that the rule S-EQUIVALL is replaced with the rule S-KERNELALL.

Conservativity of subtyping Our conservativity result for subtyping is relatively easy to establish:

Lemma 114 (Conservativity for subtyping)**.** If $\vdash_F \Gamma$, $\Gamma \vdash_F A$, and $\Gamma \vdash_F B$ then $\Gamma \vdash_F A \leq B$ if and only if $\Gamma \vdash A \leq B$.

Here the well-formedness conditions ensure that Γ, A and B must be respectively a valid $F_{\leq}$ environment, and valid $F_{\leq}$ types. That is they cannot contain recursive types (or record types). Therefore the lemma states that for environments and types without recursive types, the two subtyping relations (for $F_{\leq}$ and $F^{\mu}_{\leq}$) are equivalent, accepting the same statements. The only hurdle is that to establish the correspondence between rule S-EQUIVALL and the rule S-KERNELALL in kernel $F_{\leq}$, we need the antisymmetry property for kernel $F_{\leq}$ [13].

Conservativity of typing It is straightforward to obtain part of the conservativity result from a typing relation in $F_{\leq}$ to a typing relation in $F^{\mu}_{\leq}$. As for the reverse direction, the situation is more complicated. If we want to derive $\Gamma \vdash_F e : A$ from $\Gamma \vdash e : A$, when doing induction, for the subsumption case (rule TYPING-SUB), we need to guess an intermediate type. However, we do not know if it involves recursive types or not. Consider the following example:

$$\textsc{typing-sub}\ \frac{\textsc{typing-sub}\ \dfrac{\vdash \lambda x.\, x : \top \to \top \qquad \vdash \top \to \top \leq (\mu\alpha.\, \top) \to \top}{\vdash \lambda x.\, x : (\mu\alpha.\top) \to \top} \qquad \vdash (\mu\alpha.\, \top) \to \top \leq \top}{\vdash \lambda x.\, x : \top}$$

Although $\vdash \lambda x.\, x : \top$ do not involve recursive types, the typing subderivations can contain recursive types. As a result, the induction hypothesis cannot be applied.

This problem can be addressed by employing the algorithmic formulation of $F^{\mu}_{\leq}$, shown in Chapter 8.2.4. With algorithmic typing, we can have more precise information about the types of an expression, since algorithmic typing always gives the minimum type. Therefore, it can be proved that for expressions that do not use fold/unfold constructors, their minimum

types do not contain recursive types as well. Conservativity for algorithmic typing is proved as follows:

Lemma 115. If $\vdash_F \Gamma, \Gamma \vdash_F A$ and $\vdash_F e$ then $\Gamma \vdash_a e : A$ implies $\Gamma \vdash_F e : A$.

Now, given a typing relation $\Gamma \vdash e : A$ in $F^{\mu}_{\leq}$, we first use the minimum typing property (Theorem 103) to obtain its minimum type B such that $\Gamma \vdash_a e : B$ and $\Gamma \vdash B \leq A$. Applying Lemma 115 and Lemma 114, we complete the the conservativity proof for the declarative version of $F^{\mu}_{\leq}$.

Theorem 116 (Conservativity)**.** If $\vdash_F \Gamma, \Gamma \vdash_F A$ and $\vdash_F e$ then $\Gamma \vdash_F e : A$ if and only if $\Gamma \vdash e : A$.

8.4 Lower and Upper Bounded Quantification

In this section we introduce an extension of $F^{\mu}_{\leq}$, called $F^{\mu}_{\leq\geq}$, with lower bounded quantification and a bottom type. While upper bounded quantification has received a lot of attention in previous research, lower bounded quantification for an $F_{\leq}$-like language is much less explored, though it appears on a few works [91, 8]. We follow the same approach as Oliveira et al. [91], where their $F_{\leq}$ extension allows type variables to have either a lower bound or an upper bound, but not both bounds at once. As discussed in Chapter 8.1, this extension enables further applications, such as a form of extensible encodings of datatypes. We have proved all the same results for $F^{\mu}_{\leq\geq}$ that were proved for $F^{\mu}_{\leq}$, including type soundness, decidability, transitivity and conservativity over $F_{\leq}$.

8.4.1 The $F^{\mu}_{\leq\geq}$ Calculus

The syntax of types, expressions, values and contexts for the extended $F^{\mu}_{\leq\geq}$ calculus is shown below. The main novelties are that bottom types and lower bounded quantification are introduced. The syntactic additions are highlighted in gray.

Types	$A, B, \ldots$	$::=$	$\mathsf{nat} \mid \top \mid \bot \mid A_1 \to A_2 \mid \alpha \mid \mu\alpha.\, A \mid A^{\alpha}$
			$\mid \forall(\alpha \leq A).\, B \mid \forall(\alpha \geq A).\, B \mid \{l_i : A_i\ ^{i \in 1 \cdots n}\}$
Expressions	e	$::=$	$x \mid \mathsf{i} \mid e_1\ e_2 \mid \lambda x : A.\, e \mid e\ A \mid \Lambda(\alpha \leq A).\, e \mid \Lambda(\alpha \geq A).\, e$
			$\mid \mathsf{unfold}\ [A]\ e \mid \mathsf{fold}\ [A]\ e \mid \{l_i = e_i\ ^{i \in 1 \cdots n}\} \mid e.l$
Values	v	$::=$	$\mathsf{i} \mid \lambda x : A.\, e \mid \mathsf{fold}\ [A]\ v \mid \Lambda(\alpha \leq A).\, e \mid \Lambda(\alpha \geq A).\, e$
			$\mid \{l_i = v_i\ ^{i \in 1 \cdots n}\}$
Contexts	Γ	$::=$	$\cdot \mid \Gamma, \alpha \leq A \mid \Gamma, \alpha \geq A \mid \Gamma, x : A$

Subtyping, typing and reduction. Similarly to Chapter 8.2, the well-formedness for the additional bottom types and universal types with lower bounds are standard. The well-formed types, subtyping, typing and reduction rules for $F^{\mu}_{\leq\geq}$ are shown in Figure 8.5, Figure 8.6, Figure 8.7 and Figure 8.8, respectively. Compared with $F^{\mu}_{\leq}$, we add rule S-BOT, S-VARTRANSLB and S-EQUIVALLLB, which are the dual forms of rule S-TOP, S-VARTRANS and S-EQUIVALL, respectively. The rule TYPING-TAPPLB and TYPING-TABSLB are the dual forms of rule TYPING-TAPP and

$\boxed{\Gamma \vdash A}$ *(Well-formed Type)*

$$\textsc{wft-nat}\ \frac{}{\Gamma \vdash \mathsf{nat}} \qquad \textsc{wft-Top}\ \frac{}{\Gamma \vdash \top} \qquad \textsc{wft-Bot}\ \frac{}{\Gamma \vdash \bot} \qquad \textsc{wft-fvar}\ \frac{\alpha \leq A \in \Gamma}{\Gamma \vdash \alpha} \qquad \textsc{wft-all}\ \frac{\Gamma \vdash A \qquad \Gamma, \alpha \leq A \vdash B}{\Gamma \vdash \forall(\alpha \leq A).\ B}$$

$$\textsc{wft-alllb}\ \frac{\Gamma \vdash A \qquad \Gamma, \alpha \geq A \vdash B}{\Gamma \vdash \forall(\alpha \geq A).\ B} \qquad \textsc{wft-arrow}\ \frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \to A_2} \qquad \textsc{wft-rec}\ \frac{\Gamma, \alpha \vdash A}{\Gamma \vdash \mu\alpha.\ A} \qquad \textsc{wft-flabel}\ \frac{\Gamma \vdash A}{\Gamma \vdash A^{\alpha}}$$

$$\textsc{wft-rcd}\ \frac{\Gamma \vdash A_i \qquad \text{for each i}}{\Gamma \vdash \{l_i : A_i\ ^{i \in 1 \cdots n}\}}$$

Figure 8.5: The well-formed types for $F^{\mu}_{\leq\geq}$.

TYPING-TABS for typing, respectively. The rule STEP-TABSLB rule is the dual form of rule STEP-TABS for reduction.

The new subtyping relation is reflexive and transitive:

Theorem 117 (Reflexivity for $F^{\mu}_{\leq\geq}$)**.** If $\vdash \Gamma$ and $\Gamma \vdash A$ then $\Gamma \vdash A \leq A$.

Theorem 118 (Transitivity for $F^{\mu}_{\leq\geq}$)**.** If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$.

Structural folding and lower bounded quantification. The structural folding rule TYPING-SFOLD on recursive types has already been shown for $F^{\mu}_{\leq}$. Note that this rule is not strictly necessary for $F^{\mu}_{\leq}$, because a recursive type can only be a subtype of another recursive type or the $\top$ type. Thus the effect of structural folding in $F^{\mu}_{\leq}$, can be subsumed by other subtyping/typing rules. Perhaps for this reason, Abadi et al. [3] have only considered a structural unfolding rule.

However, in $F^{\mu}_{\leq\geq}$, a recursive type can also be a subtype of a type variable. In this case, the structural folding rule can give the desired typings to the $\mathsf{Add}_{\forall}$ constructors of the Exp_1 and Exp_2 datatypes that we have presented in Chapter 8.1.5, while the standard folding rule cannot. The rule TYPING-SFOLD has the same form in $F^{\mu}_{\leq\geq}$ as in $F^{\mu}_{\leq}$. Therefore we believe that the structural folding rule that we have proposed, together with the structural unfolding lemma in the metatheory, is general.

Type soundness. Our type soundness proof for $F^{\mu}_{\leq\geq}$ is standard, thus we have:

Theorem 119 (Preservation for $F^{\mu}_{\leq\geq}$)**.** If $\vdash e : A$ and $e \hookrightarrow e'$ then $\vdash e' : A$.

Theorem 120 (Progress for $F^{\mu}_{\leq\geq}$)**.** If $\vdash e : A$ then e is a value or exists $e', e \hookrightarrow e'$.

8.4.2 Metatheory of $F^{\mu}_{\leq\geq}$

The addition of lower bounded quantification and the bottom type create some difficulties in the metatheory of $F^{\mu}_{\leq\geq}$. The proof strategies for the unfolding lemma, decidability and conservativity, as we showed in the Chapter 8.3 for $F^{\mu}_{\leq}$, require some adjustments. In the following, we describe how to overcome the difficulties.

$\boxed{\Gamma \vdash A \leq B}$ *(Subtyping)*

$$\textsc{S-NAT}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{nat} \leq \mathsf{nat}} \qquad \textsc{S-TOP}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash A \leq \top} \qquad \textsc{S-VAR}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \alpha}{\Gamma \vdash \alpha \leq \alpha} \qquad \textsc{S-ARROW}\ \frac{\Gamma \vdash B_1 \leq A_1 \qquad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2}$$

$$\textsc{S-FREC}\ \frac{\Gamma, \alpha \leq \top \vdash [\alpha \mapsto A^{\alpha}]\, A \leq [\alpha \mapsto B^{\alpha}]\, B}{\Gamma \vdash \mu\alpha.\, A \leq \mu\alpha.\, B} \qquad \textsc{S-FLABEL}\ \frac{\Gamma \vdash A \leq B}{\Gamma \vdash A^{\alpha} \leq B^{\alpha}}$$

$$\textsc{S-EQUIVALL}\ \frac{\Gamma \vdash A_1 \leq A_2 \qquad \Gamma \vdash A_2 \leq A_1 \qquad \Gamma, \alpha \leq A_2 \vdash B \leq C}{\Gamma \vdash \forall(\alpha \leq A_1).\, B \leq \forall(\alpha \leq A_2).\, C} \qquad \textsc{S-VARTRANS}\ \frac{\Gamma \vdash B \leq A \qquad \alpha \leq B \in \Gamma}{\Gamma \vdash \alpha \leq A}$$

$$\textsc{S-RCD}\ \frac{\vdash \Gamma \qquad \Gamma \vdash \{k_j : A_j\ ^{j \in 1 \cdots m}\} \qquad \{l_i\ ^{i \in 1 \cdots n}\} \subseteq \{k_j\ ^{j \in 1 \cdots m}\} \qquad k_j = l_i \text{ implies } \Gamma \vdash A_j \leq B_i}{\Gamma \vdash \{k_j : A_j\ ^{j \in 1 \cdots m}\} \leq \{l_i : B_i\ ^{i \in 1 \cdots n}\}}$$

$$\textsc{S-BOT}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\Gamma \vdash \bot \leq A} \qquad \textsc{S-VARTRANSLB}\ \frac{\Gamma \vdash A \leq B \qquad \alpha \geq B \in \Gamma}{\Gamma \vdash A \leq \alpha}$$

$$\textsc{S-EQUIVALLLB}\ \frac{\Gamma \vdash A_1 \leq A_2 \qquad \Gamma \vdash A_2 \leq A_1 \qquad \Gamma, \alpha \geq A_2 \vdash B \leq C}{\Gamma \vdash \forall(\alpha \geq A_1).\, B \leq \forall(\alpha \geq A_2).\, C}$$

Figure 8.6: The subtyping rules for $F^{\mu}_{\leq\geq}$.

Unfolding Lemma A type system that simultaneously allows introducing lower and upper bounded type variables will break the proof of the unfolding lemma shown in Chapter 8.3.1. The interaction of lower bounds and upper bounds invalidates the inversion lemma for rule S-VARTRANS (Lemma 109). The inversion lemma holds when the bounds in the context can only have one direction. However, when we have both kinds of bounds in the context, a counter-example can be found as follows:

$$x \leq \top, y \geq x \vdash x \leq y \quad \not\Longrightarrow \quad x \leq \top, y \geq x \vdash \top \leq y$$

The existing generalized unfolding lemma for $F^{\mu}_{\leq}$ relied on the inversion properties for every subtyping rule. It is because in the generalized unfolding lemma, we have 2 subtyping statements related to type A and B. However, when a type system involves subtyping rules where the inversion properties do not hold, such as intersection types or bounded quantification with both upper bounds and lower bounds (as $F^{\mu}_{\leq\geq}$), the current generalized unfolding lemma cannot be applied.

By observing the form of the generalized unfolding lemma (Lemma 108), we can see that, when we apply the nominal unfolding rules for iso-recursive types, the conditions (1) and (2) raised by the goals implicitly subsume the premise (1). That is because 1-time unfoldings can be derivable from general nominal unfoldings.

$\boxed{\Gamma \vdash e : A}$ *(Typing)*

$$\textsc{typing-nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbf{i} : \mathsf{nat}} \qquad \textsc{typing-var}\ \frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{typing-sub}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}$$

$$\textsc{typing-abs}\ \frac{\Gamma, x : A_1 \vdash e : A_2}{\Gamma \vdash \lambda x : A_1.\ e : A_1 \to A_2} \qquad \textsc{typing-sfold}\ \frac{\Gamma \vdash e : [\alpha \mapsto B]\ A \qquad \Gamma \vdash \mu\alpha.\ A \leq B}{\Gamma \vdash \mathsf{fold}\ [B]\ e : B}$$

$$\textsc{typing-sunfold}\ \frac{\Gamma \vdash e : A \qquad \Gamma \vdash A \leq \mu\alpha.\ B}{\Gamma \vdash \mathsf{unfold}\ [A]\ e : [\alpha \mapsto A]\ B} \qquad \textsc{typing-app}\ \frac{\Gamma \vdash e_1 : A_1 \to A_2 \qquad \Gamma \vdash e_2 : A_1}{\Gamma \vdash e_1\ e_2 : A_2}$$

$$\textsc{typing-proj}\ \frac{\Gamma \vdash e : \{l_i : A_i\ ^{i \in 1 \cdots n}\}}{\Gamma \vdash e.l_i : A_i} \qquad \textsc{typing-tabs}\ \frac{\Gamma, \alpha \leq A \vdash e : B}{\Gamma \vdash \Lambda(\alpha \leq A).\ e : \forall(\alpha \leq A).\ B}$$

$$\textsc{typing-tapp}\ \frac{\Gamma \vdash e : \forall(\alpha \leq B_1).\ B_2 \qquad \Gamma \vdash A \leq B_1}{\Gamma \vdash e\ A : [\alpha \mapsto A]\ B_2} \qquad \textsc{typing-rcd}\ \frac{\text{for each } i \quad \Gamma \vdash e_i : A_i}{\Gamma \vdash \{l_i = e_i\ ^{i \in 1 \cdots n}\} : \{l_i : A_i\ ^{i \in 1 \cdots n}\}}$$

$$\textsc{typing-tapplb}\ \frac{\Gamma \vdash e : \forall(\alpha \geq B_1).\ B_2 \qquad \Gamma \vdash B_1 \leq A}{\Gamma \vdash e\ A : [\alpha \mapsto A]\ B_2} \qquad \textsc{typing-tabslb}\ \frac{\Gamma, \alpha \geq A \vdash e : B}{\Gamma \vdash \Lambda(\alpha \geq A).\ e : \forall(\alpha \geq A).\ B}$$

Figure 8.7: The typing rules for $F^{\mu}_{\leq\geq}$.

With such an important observation, we merge two subtyping statements related to type A and B into one. To accommodate this change, the context is also needed to refine. Formally, the generalized unfolding lemma for $F^{\mu}_{\leq\geq}$ is:

Lemma 121 (Generalized unfolding lemma for $F^{\mu}_{\leq\geq}$)**.** If

1. $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2 \vdash A$ and $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2 \vdash B$;
2. $G \vdash [\alpha \mapsto C^{\alpha}]\ A \leq [\alpha \mapsto D^{\alpha}]\ B$
3. G differs from $\Gamma_1,\ \alpha \leq \top,\ \Gamma_2[\alpha \mapsto S^{\alpha}]$ only in the components labeled by α, where S^{α} can be replaced by T^{α} that satisfies $\Gamma_2 \vdash \mu\alpha.\ S \leq \mu\alpha.\ T$ and $\Gamma_2 \vdash \mu\alpha.\ T \leq \mu\alpha.\ S$

then

1. $\Gamma_1 \vdash \mu\alpha.\ C \leq \mu\alpha.\ S$ and $\Gamma_1 \vdash \mu\alpha.\ S \leq \mu\alpha.\ D$ implies $\Gamma_1,\ \Gamma_2[\alpha \mapsto \mu\alpha.\ S] \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$;
2. $\Gamma_1 \vdash \mu\alpha.\ D \leq \mu\alpha.\ S$ and $\Gamma_1 \vdash \mu\alpha.\ S \leq \mu\alpha.\ C$ implies $\Gamma_1,\ \Gamma_2[\alpha \mapsto \mu\alpha.\ S] \vdash [\alpha \mapsto \mu\alpha.\ C]\ A \leq [\alpha \mapsto \mu\alpha.\ D]\ B$;

Proof. Do induction on premise (2) and then premise (1), respectively. □

Lemma 122 (Unfolding lemma for $F^{\mu}_{\leq\geq}$)**.** If $\Gamma \vdash \mu\alpha.\ A \leq \mu\alpha.\ B$ then $\Gamma \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\alpha \mapsto \mu\alpha.\ B]\ B$.

$\boxed{e_1 \hookrightarrow e_2}$ *(Reduction)*

STEP-BETA
$$\frac{}{(\lambda x : A.\ e_1)\ v_2 \hookrightarrow [x \mapsto v_2]\ e_1}$$

STEP-APPL
$$\frac{e_1 \hookrightarrow e_1'}{e_1\ e_2 \hookrightarrow e_1'\ e_2}$$

STEP-APPR
$$\frac{e_2 \hookrightarrow e_2'}{v_1\ e_2 \hookrightarrow v_1\ e_2'}$$

STEP-FLD
$$\frac{}{\mathsf{unfold}\ [A]\ (\mathsf{fold}\ [B]\ v) \hookrightarrow v}$$

STEP-UNFOLD
$$\frac{e \hookrightarrow e'}{\mathsf{unfold}\ [A]\ e \hookrightarrow \mathsf{unfold}\ [A]\ e'}$$

STEP-FOLD
$$\frac{e \hookrightarrow e'}{\mathsf{fold}\ [A]\ e \hookrightarrow \mathsf{fold}\ [A]\ e'}$$

STEP-TAPP
$$\frac{e_1 \hookrightarrow e_2}{e_1\ A \hookrightarrow e_2\ A}$$

STEP-TABS
$$\frac{}{(\Lambda(\alpha \leq A).\ e)\ B \hookrightarrow [\alpha \mapsto B]\ e}$$

STEP-PROJ
$$\frac{e \hookrightarrow e'}{e.l_j \hookrightarrow e'.l_j}$$

STEP-PROJRCD
$$\frac{}{\{l_i = v_i\ ^{i \in 1 \cdots n}\}.l_j \hookrightarrow v_j}$$

STEP-TABSLB
$$\frac{}{(\Lambda(\alpha \geq A).\ e)\ B \hookrightarrow [\alpha \mapsto B]\ e}$$

STEP-RCD
$$\frac{e_j \hookrightarrow e_j'}{\{l_i = v_i\ ^{i \in 1 \cdots j-1},\ l_j = e_j,\ l_k = e_k\ ^{k \in j+1 \cdots n}\} \hookrightarrow \{l_i = v_i\ ^{i \in 1 \cdots j-1},\ l_j = e_j',\ l_k = e_k\ ^{k \in j+1 \cdots n}\}}$$

Figure 8.8: The reduction rules for $F^{\mu}_{\leq\geq}$.

Decidability The interaction between bottom types and rule S-EQUIVALL breaks the measure-based decidability proof in Chapter 8.3.2. The bottom type in $F^{\mu}_{\leq}$ brings a new form of equivalent types: when $\alpha \leq \bot \in \Gamma$, one can derive that $\Gamma \vdash \alpha \leq \bot$ and $\Gamma \vdash \bot \leq \alpha$, as has been observed by Pierce [96]. Simply extending the measure function with $size_\Psi(\bot) = 1$ will not work. For type variables, the measure function will recursively look up its bound in the context, and add one to the measure of its bound, making a variable equivalent to $\bot$ to *have a larger measure* than $\bot$. Therefore, replacing two equivalent types into the abstracted type body may not produce the same measures. We can construct derivations of rule S-EQUIVALL that have a larger measure in the premise than that of the conclusion, which makes the decidability proof fail with the current measure.

A counter-example is:

$$\frac{\alpha \leq \bot,\ \beta \leq \alpha \vdash \alpha \leq \beta \qquad \alpha \leq \bot,\ \beta \leq \alpha \vdash \beta \leq \alpha \qquad \alpha \leq \bot,\ \beta \leq \alpha,\ \gamma \leq \beta \vdash A \leq B}{\alpha \leq \bot,\ \beta \leq \alpha \vdash \forall(\gamma \leq \alpha).\ A \leq \forall(\gamma \leq \beta).\ B}$$

The measure for premise (3) is:

$$size_{\alpha \mapsto 1,\ \beta \mapsto 2,\ \gamma \mapsto 3}(A) + size_{\alpha \mapsto 1,\ \beta \mapsto 2,\ \gamma \mapsto 3}(B)$$

while the measure for the goal is:

$$\begin{array}{ll} & size_{\alpha \mapsto 1,\ \beta \mapsto 2}(\forall(\gamma \leq \alpha).\ A) + size_{\alpha \mapsto 1,\ \beta \mapsto 2}(\forall(\gamma \leq \beta).\ B) \\ \hookrightarrow & size_{\alpha \mapsto 1,\ \beta \mapsto 2,\ \gamma \mapsto 2}(A) + size_{\alpha \mapsto 1,\ \beta \mapsto 2,\ \gamma \mapsto 3}(B) \end{array}$$

which is less than the measure of premise (3).

We solve this issue by replacing all the type variables who are a subtype of bottom by bottom. By such way, the subtyping relation $\alpha \leq \bot,\ \beta \leq \alpha \vdash \forall(\gamma \leq \alpha).\ A \leq \forall(\gamma \leq \beta).\ B$ becomes

$$\alpha \leq \bot,\ \beta \leq \bot \vdash \forall(\gamma \leq \bot).\ A \leq \forall(\gamma \leq \bot).\ B$$

and the measure works again.

This idea can be implemented by modifying the measure function to identify upper/lower bounded variables that are equivalent to bottom/top types. Every time that we compute the size for a variable bounded by an upper bound, we firstly recursively check whether it is a synonym for bottom, and return the size of bottom if it is. This can be implemented by an extension to the measure context Ψ. Dually, when we compute the size for a variable bounded by a lower bound, we recursively check if it is the supertype of top. With this change the measures work, and we can prove decidability.

Therefore, the context is redefined as $\Psi := \cdot \mid \Psi, \alpha \mapsto i \mid \Psi, \alpha \mapsto \bot \mid \Psi, \alpha \mapsto \top$, which is used to store the measure of variables or indicate them as upper bounded by $\bot$ or lower bounded by $\top$. Then, a measure function $size_\Psi(A)$ is redefined as:

$$
\begin{array}{lcl}
size_\Psi(\mathsf{nat}) & = & 1 \\
size_\Psi(\top) & = & 1 \\
size_\Psi(\bot) & = & 1 \\
size_\Psi(A \to B) & = & 1 + size_\Psi(A) + size_\Psi(B) \\
size_\Psi(A^\alpha) & = & 1 + size_\Psi(A) \\
size_\Psi(\alpha) & = & 1 + \begin{cases} i & \alpha \mapsto i \in \Psi \\ 0 & \text{otherwise} \end{cases} \\
size_\Psi(\forall(\alpha \leq A).\ B) & = & 1 + \begin{cases} 1 + size_{\Psi,\alpha\mapsto\bot}(B) & isBot_\Psi(A) \\ size_\Psi(A) + size_{\Psi,\alpha\mapsto size_\Psi(A)}(B) & \text{otherwise} \end{cases} \\
size_\Psi(\forall(\alpha \geq A).\ B) & = & 1 + \begin{cases} 1 + size_{\Psi,\alpha\mapsto\top}(B) & isTop_\Psi(A) \\ size_\Psi(A) + size_{\Psi,\alpha\mapsto size_\Psi(A)}(B) & \text{otherwise} \end{cases} \\
size_\Psi(\mu\alpha.\ A) & = & \text{let } i := size_{\Psi,\alpha\mapsto 1}(A) \text{ in } 1 + size_{\Psi,\alpha\mapsto i}(A)
\end{array}
$$

$$
\begin{array}{lll}
isBot_\Psi(\bot) & = \mathsf{true} & isTop_\Psi(\top) = \mathsf{true} \\
isBot_{\Psi,\alpha\mapsto\bot}(\alpha) & = \mathsf{true} & isTop_{\Psi,\alpha\mapsto\top}(\alpha) = \mathsf{true} \\
isBot_{\Psi,\beta\mapsto\_}(\alpha) & = isBot_\Psi(\alpha) \text{ if } \alpha \neq \beta & isTop_{\Psi,\beta\mapsto\_}(\alpha) = isTop_\Psi(\alpha) \text{ if } \alpha \neq \beta \\
\text{otherwise } isBot_\Psi(A) & = \mathsf{false} & \text{otherwise } isTop_\Psi(A) = \mathsf{false}
\end{array}
$$

The *isTop* and *isBot* functions basically use the information in the measure context Ψ to check whether the bound type A is equivalent to $\top$ or $\bot$. If so, when the variable bounded by A is looked up in the context, it will have a measure of 1. In this way, we retain the important property that equivalent types have the same measure.

$\boxed{\Gamma \vdash A \Uparrow B}$ *(Upper Exposure)*

$$\textsc{xa-promote}\ \frac{\alpha \leq A \in \Gamma \qquad \Gamma \vdash A \Uparrow B}{\Gamma \vdash \alpha \Uparrow B} \qquad \textsc{xa-upinv}\ \frac{\alpha \geq A \in \Gamma}{\Gamma \vdash \alpha \Uparrow \alpha} \qquad \textsc{xa-up}\ \frac{A \text{ is not type variable}}{\Gamma \vdash A \Uparrow A}$$

$\boxed{\Gamma \vdash A \Downarrow B}$ *(Lower Exposure)*

$$\textsc{xa-promotedown}\ \frac{\alpha \geq A \in \Gamma \qquad \Gamma \vdash A \Downarrow B}{\Gamma \vdash \alpha \Downarrow B} \qquad \textsc{xa-downinv}\ \frac{\alpha \leq A \in \Gamma}{\Gamma \vdash \alpha \Downarrow \alpha} \qquad \textsc{xa-downward}\ \frac{A \text{ is not type variable}}{\Gamma \vdash A \Downarrow A}$$

$\boxed{\Gamma \vdash_a e : A}$ *(Algorithmic Typing)*

$$\textsc{atyp-appbot}\ \frac{\Gamma \vdash_a e_1 : A \qquad \Gamma \vdash A \Uparrow \bot \qquad \Gamma \vdash_a e_2 : A_2}{\Gamma \vdash_a e_1\ e_2 : \bot} \qquad \textsc{atyp-tappbot}\ \frac{\Gamma \vdash_a e : B \qquad \Gamma \vdash B \Uparrow \bot \qquad \Gamma \vdash A}{\Gamma \vdash_a e\ A : \bot}$$

$$\textsc{atyp-sunfoldbot}\ \frac{\Gamma \vdash_a e : A \qquad \Gamma \vdash B \Uparrow \bot \qquad \Gamma \vdash A \leq B}{\Gamma \vdash_a \mathsf{unfold}\ [B]\ e : \bot} \qquad \textsc{atyp-sfoldtop}\ \frac{\Gamma \vdash_a e : A \qquad \Gamma \vdash C \Downarrow \top \qquad \Gamma \vdash C}{\Gamma \vdash_a \mathsf{fold}\ [C]\ e : \top}$$

Figure 8.9: The new exposure function and additional algorithmic typing rules for $F^{\mu}_{\leq\geq}$.

Theorem 123 (Decidability for $F^{\mu}_{\leq\geq}$)**.** $F^{\mu}_{\leq\geq}$ is decidable.

Conservativity The proof of conservativity for $F^{\mu}_{\leq\geq}$ follows the same pattern as the proof for $F^{\mu}_{\leq}$. To prove conservativity of typing, we need the help of the algorithmic typing rules to obtain the minimum type of an $F_{\leq}$ term. However, the introduction of bottom types requires us to add several new algorithmic typing rules, since in the declarative system, one can always use the subsumption rule to transform a term with type $\bot$ to any function type or universal type, and apply it to any argument, as has been observed by Pierce [96]. We also develop a similar treatment for recursive types. Moreover, the meanings of the two exposure functions also need to be refined. For example, the upper exposure function ($\Uparrow$) is now used to find the least non-*upper-bounded*-variable upper bound in the context, so it will return the variable itself if the variable is lower bounded. The complete set of $F^{\mu}_{\leq\geq}$ algorithmic typing rules can be found at Figure 8.9.

Theorem 124 (Conservativity for $F^{\mu}_{\leq\geq}$)**.** If $\vdash_F \Gamma, \Gamma \vdash_F A$ and $\vdash_F e$ then $\Gamma \vdash_F e : A$ if and only if $\Gamma \vdash e : A$.

8.5 Mechanized Proofs

The folder *coq_fsub* includes all the Coq proofs about $F^{\mu}_{\leq}$ and $F^{\mu}_{\leq\geq}$ in this chapter. The folder *coq_fsub/coq_fsub_main* contains all the definitions and proofs about $F^{\mu}_{\leq}$ and the folder *coq_fsub/coq_fsub_all* contains all the definitions and proofs about $F^{\mu}_{\leq\geq}$.

Table 8.1: Paper-to-proofs correspondence guide in Chapter 8.

Definition	File	Name in Coq	Notation
$F^{\mu}_{\leq}$, in folder *coq_fsub/coq_fsub_main*			
Well-formed Type (Figure 8.1)	Rules.v	WF E A	$\vdash \Gamma$
Subtyping (Figure 8.1)	Rules.v	sub E A B	$\vdash \Gamma \vdash A \leq B$
Typing (Figure 8.2)	Rules.v	typing E e A	$\vdash \Gamma \vdash e : A$
Reduction (Figure 8.3)	Rules.v	step e1 e2	$e_1 \hookrightarrow e_2$
Upper Exposure (Figure 8.4)	AlgoTyping.v	exposure E A B	$\Gamma \vdash A \Uparrow B$
Lower Exposure (Figure 8.4)	AlgoTyping.v	exposure2 E A B	$\Gamma \vdash A \Downarrow B$
Algorithmic Typing (Figure 8.4)	AlgoTyping.v	algo_typing E e A	$\Gamma \vdash_a e : A$
$F^{\mu}_{\leq\geq}$, in folder *coq_fsub/coq_fsub_all*			
Well-formed Type (Figure 8.5)	Rules.v	WF E A	$\vdash \Gamma$
Subtyping (Figure 8.6)	Rules.v	sub E A B	$\vdash \Gamma \vdash A \leq B$
Typing (Figure 8.7)	Rules.v	typing E e A	$\vdash \Gamma \vdash e : A$
Reduction (Figure 8.8)	Rules.v	step e1 e2	$e_1 \hookrightarrow e_2$
Upper Exposure (Figure 8.9)	AlgoTyping.v	exposure E A B	$\Gamma \vdash A \Uparrow B$
Lower Exposure (Figure 8.9)	AlgoTyping.v	exposure2 E A B	$\Gamma \vdash A \Downarrow B$
Algorithmic Typing (Figure 8.9)	AlgoTyping.v	algo_typing E e A	$\Gamma \vdash_a e : A$

8.5.1 Definitions

All the definitions of $F^{\mu}_{\leq}$ can be found in files *coq_fsub_main/Rules.v*, and all the definitions of $F^{\mu}_{\leq\geq}$ showed in Chapter 8.4 can be found in files *coq_fsub_all/Rules.v*. Table 8.1 shows the correspondence of definitions between the paper and the Coq artifacts.

8.5.2 Lemmas and theorems

Table 8.2 shows the descriptions for all the proof scripts in Chapter 9. It contains some important lemmas and theorems for $F^{\mu}_{\leq}$. The lemmas and theorems for $F^{\mu}_{\leq\geq}$, with the same name and structure as those for $F^{\mu}_{\leq}$ in the folder *coq_fsub_main*, can be found at the folder *coq_fsub_dual*.

Table 8.2: Descriptions for the proof scripts in Chapter 8.

Theorems	Description	Files	Name in Coq
	$F^{\mu}_{\leq}$, in folder *coq_fsub/coq_fsub_main*		
Theorem 96	Reflexivity	Reflexivity.v	Reflexivity
Theorem 97	Transitivity	Transitivity.v	sub_transitivity
Lemma 98	Unfolding Lemma	Unfolding.v	unfolding_lemma
Theorem 100	Preservation	Preservation.v	preservation
Theorem 101	Progress	Progress.v	progress
Theorem 102	Soundness of algorithmic rules	AlgoTyping.v	typing_algo_sound
Theorem 103	Completeness of algorithmic rules	AlgoTyping.v	minimum_typing
Theorem 111	Decidability	Decidability.v	decidability
Theorem 114	Conservativity for Subtyping	Conservativity.v	sub_conserv
Theorem 116	Conservativity	Conservativity.v	typing_conserv
	$F^{\mu}_{\leq\geq}$, in folder *coq_fsub/coq_fsub_all*		
Theorem 117	Reflexivity	Reflexivity.v	Reflexivity
Theorem 118	Transitivity	Transitivity.v	sub_transitivity
Theorem 119	Preservation	Preservation.v	preservation
Theorem 120	Progress	Progress.v	progress
Lemma 122	Unfolding Lemma	Unfolding.v	unfolding_lemma
Theorem 123	Decidability	Decidability.v	decidability
Theorem 124	Conservativity	Conservativity.v	typing_conserv

Part IV

Epilogue

Chapter 9

Related Work

Throughout the thesis, we have already reviewed some of the closest related work in detail. In this chapter, we will discuss other related work.

9.1 Subtyping Recursive Types

9.1.1 Iso-recursive Amber rules

In Chapter 2.1.3, we discussed Amadio and Cardelli [5]'s work on recursive types. Their work is about equi-recursive types, which is enabled by a very expressive equivalence relation used in their reflexivity rule. Much of the follow-up work has employed a much weaker alpha-equivalence relation in the Amber rules, leading to an iso-recursive formulation of subtyping.

With respect to the metatheory of iso-recursive subtyping with the Amber rules, Bengtson et al. [18]'s work is the closest to ours. They manually proved a full set of type safety properties, including the transitivity lemma for subtyping and the unfolding lemma (as a part of their inversion lemma). The transitivity lemma, "*perhaps the most difficult*" statement in their work, is proven with a complex inductive argument. For example, a subtyping chain of type variables, $\alpha_1 \leq \alpha_2 \leq \alpha_3$, is accepted by their transitivity statement, by means of adapting variable bindings in the contexts accordingly:

$$\frac{\Gamma[\alpha_1 \leq \alpha_2] \vdash \alpha_1 \leq \alpha_2 \qquad \Gamma[\alpha_2 \leq \alpha_3] \vdash \alpha_2 \leq \alpha_3}{\Gamma[\alpha_1 \leq \alpha_3] \vdash \alpha_1 \leq \alpha_3}$$

In other words, the subtyping judgments of their transitivity statement (used for their proof) do not share the same context, which subtly captures the nature of context elements ($\alpha \leq \beta$) in the Amber rules. Such technique involving inconsistent contexts is an uncommon practice, and it complicates the proof. Backes et al. [12] attempted to formalize this transitivity proof in Coq, but they failed, stating that: "*The soundness of the Amber rule (Sub Rec) is hard to prove syntactically – in particular proving the transitivity of subtyping in the presence of the Amber rule requires a very complicated inductive argument, which only works for "executable" environments*".

Many other works avoid some of the complexity in the metatheory of the Amber rules by employing a declarative subtyping relation with transitivity built-in [83, 38, 56, 103, 1].

However, this leaves open the question of how to obtain a sound and complete algorithmic formulation, which as discussed in Chapter 2.1.3 and 5.2, is non-trivial. Chugh [44] observes the lack of some desirable properties (such as decidability) and difficulties of implementing languages modelling foundational aspects of Object-Oriented Programming when employing calculi with equi-recursive types. To address those difficulties he proposes a source calculus with iso-recursive types using the Amber rules, which enables decidability. He does not discuss transitivity of subtyping for the source calculus. Type-safety of the source calculus is shown via an elaboration into a target calculus with equi-recursive types and *f-bounded polymorphism* [29]. In general, those works employ elaboration and/or coercive subtyping, which leads to an alternative way to prove type-safety, and transitivity is either built-in or not discussed. In contrast, our metatheory comes with transitivity proofs, as well as a direct operational semantics for a calculus with iso-recursive types.

9.1.2 Complete iso-recursive subtyping

Ligatti et al. [84] propose an improvement to the Amber rules for iso-recursive subtyping. They observe that the Amber rules are sound, but incomplete with respect to type-safety. Besides the complications due to the presence of a reflexivity rule, they find that a source of incompleteness of the Amber rules comes from complications with recursive type unrolling. The two rules for subtyping recursive types employed by Ligatti et al. [84] are:

$$\frac{S, \mu\alpha.\ A \leq \mu\beta.\ B \vdash [\alpha \mapsto \mu\alpha.\ A]\ A \leq [\beta \mapsto \mu\beta.\ B]\ B}{S \vdash \mu\alpha.\ A \leq \mu\beta.\ B}\text{L17-REC1}$$

$$\frac{(\mu\alpha.\ A \leq \mu\beta.\ B) \in S}{S \vdash \mu\alpha.\ A \leq \mu\beta.\ B}\text{L17-REC2}$$

The basic idea is that subtyping environments S track all subtyping relations between recursive types that have already been observed. Rule L17-REC1 is the rule that is triggered if $\mu\alpha.\ A \leq \mu\beta.\ B$ has not been observed yet. In that case $\mu\alpha.\ A \leq \mu\beta.\ B$ is simply added to the environment and the recursive type variables are directly replaced by the recursive types in the bodies. In rule L17-REC2, if $\mu\alpha.\ A \leq \mu\beta.\ B$ is already in the environment, then we know that the two recursive types are in a subtyping relation and we can terminate. One similarity to the double unfolding and the nominal unfolding rules is that both Ligatti et al. [84]'s rules and our rules employ one substitution for each type. However, in Ligatti et al. [84]'s rules we substitute the recursive type variable with the recursive type directly, whereas in our rules we use a finite unfolding: that is we use the body of the recursive type instead. Ligatti et al. [84]'s rules are more powerful than both the Amber rules and all the rules presented in this thesis, including our declarative formulation with finite unfoldings, as well as the double and nominal unfolding rules. The simplest example that illustrates the different expressive power between our rules and the rules by Ligatti et al. [84] is perhaps $\mu\alpha.\ \alpha \leq \mu\alpha.\ (\mu\beta.\ \alpha)$. With Ligatti et al. [84]'s rules this is a valid subtyping statement, as illustrated by the following derivation:

$$\dfrac{\dfrac{\dfrac{\mu\alpha.\,\alpha \leq \mu\alpha.\,(\mu\beta.\,\alpha) \in \{\mu\alpha.\,\alpha \leq \mu\alpha.\,(\mu\beta.\,\alpha),\ \mu\alpha.\,\alpha \leq \mu\beta.\,(\mu\alpha.\,(\mu\beta.\,\alpha))\}}{\mu\alpha.\,\alpha \leq \mu\alpha.\,(\mu\beta.\,\alpha),\ \mu\alpha.\,\alpha \leq \mu\beta.\,(\mu\alpha.\,(\mu\beta.\,\alpha)) \vdash \mu\alpha.\,\alpha \leq \mu\alpha.\,(\mu\beta.\,\alpha)}\ \text{L17-REC2}}{\mu\alpha.\,\alpha \leq \mu\alpha.\,(\mu\beta.\,\alpha) \vdash \mu\alpha.\,\alpha \leq \mu\beta.\,(\mu\alpha.\,(\mu\beta.\,\alpha))}\ \text{L17-REC1}}{\vdash \mu\alpha.\,\alpha \leq \mu\alpha.\,(\mu\beta.\,\alpha)}\ \text{L17-REC1}$$

In contrast, the rules based on finite unfoldings reject such subtyping statement. For instance, here is the failed derivation with the double unfolding rules:

$$\dfrac{\dfrac{\text{Derivation fails here}}{\alpha \vdash \alpha \leq \mu\beta.\,\alpha} \qquad \dfrac{\text{Derivation fails here}}{\alpha \vdash \alpha \leq \mu\beta.\,(\mu\beta.\,\alpha)}}{\vdash \mu\alpha.\,\alpha \leq \mu\alpha.\,(\mu\beta.\,\alpha)}\ \text{SA-REC}$$

The source of the difference in terms of expressive power between our rules (as well as the Amber rules) and Ligatti et al. [84]'s rules is related to the treatment of subtyping between type variables and recursive types. In the failed derivation with the double unfolding rule we can see that the derivation fails when we encounter a subtyping statement of the form $\alpha \leq \mu\beta.\,A$. That is when we try to compare a recursive type variable with a recursive type. In both our rules and the Amber rules, such statements are always rejected, since the recursive type variables are opaque and the structure of the recursive type denoted by the type variable is not known. In some sense with the Amber rules and our rules recursive type variables act similarly to nominal types, and comparing them with a recursive type that happens to have a structurally compatible shape will fail. In Ligatti et al. [84]'s rules, because type variables are always replaced by the recursive type, the structure of the recursive type is known (or transparent) and then the subtyping rules for recursive types can be used instead. In addition to the simple example that we describe above, Ligatti et al. [84] have identified two larger examples that demonstrate that their rules can derive subtyping statements that the Amber rules cannot:

- $\mu n.\{\text{sub} : (\mu i'.\{\text{sub} : i' \to \text{unit}\}) \to \text{unit}, \text{min} : \text{unit} \to \text{int}\} \leq \mu i.\{\text{sub} : i \to \text{unit}\}$;
- $\mu a.(((\mu b.((b + \text{nat}) + a)) + \text{nat}) + a) \leq \mu c.((c + \text{real}) + c)$.

As they observe, accepting such subtyping statements does not violate type-safety. The two examples above are also rejected by our rules, for similar reasons to the simpler example above. The failure to derive such subtyping statements is expected since we proved that our rules are equivalent in terms of expressive power to the Amber rules, which reject them as well.

In addition to rules L17-REC1 and L17-REC2, some non-standard subtyping rules for *value-uninhabited* types are also needed for achieving the completeness of subtyping with respect to type safety. If a type is value-uninhabited then every expression of that type diverges. In other words, value-uninhabited types are treated as bottom types ($\bot$). As Ligatti et al. [84] explained, if we do not care about subtyping completeness with respect to type-safety, we can ignore the extra subtyping rules for value-uninhabited types, and still get additional expressive power over the Amber rules. From the point of view of type-safety, the new formulations of subtyping proposed by us are also incomplete, since they have the same expressive power as the Amber rules.

Our declarative formulation of subtyping is essentially following a syntactic approach to subtyping, whereas a formulation based on completeness with respect to type-safety is closer in spirit to *semantic subtyping* [40]. While syntactic formulations are generally less expressive, their metatheory is usually simpler, and such formulations are also generally more extensible. To achieve their goal of a complete formulation of subtyping with respect to type safety, Ligatti et al. [84] had to develop several new proof techniques to accomplish this goal. For instance one of the techniques developed in their work is induction on *failing derivations*, which requires defining an explicit relation that captures failed derivations of subtyping. A further complicating factor is the non-standard form of environments S required by rules L17-REC1 and L17-REC2, which must contain entries of the form $\mu\alpha.\ A \leq \mu\beta.\ B$. This is in contrast to our rules, which all employ standard environments with type variables only. Both of these mean that the subtyping metatheory is significantly different from conventional formulations of subtyping. In Ligatti et al. [84]'s work, most important theorems, such as transitivity or reflexivity, are proved by doing induction on failing derivations. For example, their transitivity theorem is proved via an auxiliary theorem called *strong subtyping transitivity* of the form:

$$\frac{S \vdash \tau_1 \leq \tau_3 \text{ is not derivable} \qquad \vdash \tau_1 \leq \tau_2}{\vdash \tau_2 \leq \tau_3 \text{ is not derivable}}$$

This theorem relies on the failed derivations relation and leads to a transitivity proof that is quite different from conventional transitivity proofs for subtyping. In contrast, our transitivity theorem (as well as other lemmas such as reflexivity) and proofs are standard. For instance, as we show in Theorem 6, our transitivity proof is modular in the sense that proofs for the cases of non-recursive type constructs (such as function types) are essentially the same as for a subtyping relation without recursive types. In other words the addition of recursive types using our rules has little impact on existing proofs[1]. This is not the case in Ligatti et al. [84]'s work since their rules for recursive subtyping as well as their proof techniques for showing completeness of subtyping with respect to type-safety require new proof techniques and proofs, and even new theorem statements. In addition, all our proofs have been formalized in a theorem prover, whereas Ligatti et al. [84]'s proofs have not been mechanically formalized yet.

9.1.3 Other approaches to iso-recursive subtyping

For solving the conflict between contravariant types and recursive types, Hofmann and Pierce [72] proposed an approach where only covariant types are allowed. In their subtyping rules, the inputs of function types must be the same. Later, Hosoya et al. [73] gave an algorithm to prove transitivity and type soundness, but it still relies on a complicated environment where all of the components are pairs of structural recursive types. Thus, they have extra rules for contexts to obtain enough information for the subtyping assumptions. Featherweight Java [78], is another calculus that supports a form of iso-recursive types. Although there are no specific

[1] Our locally nameless [42] based Coq proofs follow a similar style to Chargueraud's proofs for System $F_{<:}$ in https://www.chargueraud.org/softs/ln.

recursive type constructs, recursive types appear because class declarations can be recursive. An advantage of the Featherweight Java design is that recursive types are fairly easy to model, and modeling mutually recursive types is straightforward. However, structural iso-recursive types, such as those in the Amber rules, allow for nested recursive types, which are not directly supported in Featherweight Java. Featherweight Java does support mutually recursive classes, so perhaps there is some general way to support such nested recursive types via an encoding.

9.1.4 Equi-recursive subtyping

Equi-recursive subtyping has been widely used in various calculi. With equi-recursive subtyping a recursive type is equivalent to its unfoldings. Amadio and Cardelli [5]'s work provided the first theoretical foundation for equi-recursive types. Subsequent work by Brandt and Henglein [24] and Gapeyev, Levin, and Pierce [62] improved and simplified the theory of Amadio and Cardelli [5]'s study. In particular, they advocated for the use of coinduction for the metatheory of equi-recursive subtyping. Equi-recursive types play an important role in many areas. They have been employed for session types [39, 63, 64, 43], and Siek and Tobin-Hochstadt [111] applied equi-recursive types in gradual typing.

Semantic subtyping [61, 40], provides a set-theoretic point of view for type systems. In that approach, equi-recursive types and intersection types are interpreted as subsets of the model, the subtyping relation is decidable, and some important properties, such as transitivity, are derived naturally. Although semantic subtyping approaches have many advantages, they can be technically more involved, while the metatheory of syntactic formulations is simpler and generally easier to extend. Damm [54] explored a type system with equi-recursive types and intersection types. His subtyping relation is quite expressive, as it supports equi-recursive types and distributivity rules for subtyping. However, he does not follow the conventional syntactic formulations of subtyping, such as the one in this thesis or those employed in Dependent object types (DOT) [6, 108]. Instead, types are encoded as regular tree expressions/set constraints. In contrast, our formulation is more conventional and supports iso-recursive types instead.

9.1.5 Mechanical formalizations with recursive subtyping

While to our knowledge there are no mechanical formalizations with the Amber rules, there are a few works trying to formalize other variants of recursive subtyping. Closest to our work is the Coq formalization by Backes et al. [12]. They show a Coq proof for refinement types with a positive restriction for iso-recursive types. In fact, our positive subtyping formulation (Figure 5.1) is close to Backes et al. [12]'s definition. However, our definition is more general since equal types with negative recursive occurrences are considered subtypes, whereas in their formulation recursive types with negative occurrences of recursive variables are forbidden. Appel and Felty [10] gave a related Twelf proof of positive subtyping, where function types are invariant with respect to the input types of functions. Recently, based on big-step semantics,

Amin and Rompf [8] gave a formalization of DOT, which employs a special form of equi-recursive type. Danielsson and Altenkirch [55], mixes induction and coinduction for proving properties of equi-recursive subtyping in Agda.

9.2 Object Encodings

9.2.1 Bounded quantification

Bounded quantification was firstly introduced by Cardelli and Wegner [37] in the language Fun, where their kernel Fun calculus corresponds to the kernel version of $F_{\leq}$. The full variant of $F_{\leq}$ was introduced by Curien and Ghelli [53] and Cardelli et al. [35], where the subtyping for bounds are contravariant. Although full $F_{\leq}$ is powerful, subtyping is proved to be undecidable [98]. As discussed in Chapter 1.1.4 there are several attempts to add recursive types to $F_{\leq}$, such as the work by Ghelli [65], Colazzo and Ghelli [46] and Jeffrey [80]. Unfortunately, such combinations are not painless, and even the successful combinations require the significant changes for the subtyping rules. In Colazzo and Ghelli's work, there is no independent universal type, and the shape of recursive types is either $\mu\alpha.\ \forall(x \leq A).\ B$ or $\mu\alpha.\ A \to B$. The recursive variables and universal variables are distinct, resulting in changes in environments and subtyping rules. For example, the subtyping environment is defined as $\Pi := \cdot \mid \Pi, (x, y) \leq (A, B) \mid \Pi, (\alpha = A, \beta = B)$, and the rule S-VARTRANS rule of $F_{\leq}$ is changed to:

$$\frac{(x, y) \leq (A', B') \in \Pi \qquad \forall \alpha', B \neq \alpha' \qquad B \neq \top \quad B \neq y \quad \Pi \vdash A' \leq B}{\Pi \vdash x \leq B}$$

The algorithm proposed by Jeffrey is also complex, and requires major changes. Both recursive variables and the subtyping algorithm are labelled polarly, and the implementation of α-conversion is not discussed. In contrast, our subtyping rules do not change the contexts, the types are not restricted, and most importantly, we do not have to change the rules in the original $F_{\leq}$. This has the benefit that we can largely reuse the existing metatheory of the original $F_{\leq}$, and it also enables our conservativity result. While it is plausible that Jeffrey [80]'s and Colazzo and Ghelli [46]'s work for the kernel $F_{\leq}$ extensions with recursive types are conservative, this has not been proved. Furthermore, such proof is likely to be non-trivial because of the major changes introduced by equi-recursive subtyping.

Table 9.1 summarizes the results of previous work on extending $F_{\leq}$ with recursive types. *Type System* simply means whether the typing relation of the $F_{\leq}$ extension with recursive types has been studied/presented in the paper. *Type soundness* means whether, in addition to presenting the type system, the proof of type soundness for the $F_{\leq}$ extension was also done. *Conservativity* is just a specific proof or property that describes whether the new calculus preserves the behavior of the original calculus, and it is quite different from modularity. *Modularity* is simply about whether the original rules and definitions of $F_{\leq}$ are the same or they need to be modified. Modularity is helpful, but not necessary, to prove conservativity. Essentially if most rules are just the same in the original system and the extension, then most cases in the

Table 9.1: Comparison among different work on extending $F_{\leq}$ to recursive types.

	$F^{equi}_{\leq}$ [65]	Kernel $F^{equi}_{\leq}$ [46]	$F^{equi}_{\leq}$ [80]	Kernel $F^{equi}_{\leq}$ [80]	$F^{iso}_{\leq}$ [3]	$F^{\mu}_{\leq}$ Chapter 8
Transitivity	×	✓	✓	✓	built-in	✓
Decidability		✓	✓̸	✓		✓
Conservativity	×		×			✓
Type System					✓	✓
Algorithmic Type System						✓
Type Soundness						✓
Modularity	×	×	×	×		✓
Mechanized Proofs	×	×	×	×	×	✓

Note: A × symbol denotes a negative result (the property or feature does not hold). A ✓ denotes a positive result, while ✓̸ denotes a partial result (such as semi-decidability). Whitespace denotes that the property/feature has not been studied or it is unknown.

proof are quite trivial. However, even if the definitions are changed, it may still be possible to prove conservativity. It is just that the proof may be significantly harder.

The proofs in all the 4 systems with equi-recursive types are complex because of the strong recursion. Adding equi-recursive subtyping requires major changes in existing definitions and proofs compared to $F_{\leq}$, making most of the existing metatheory on $F_{\leq}$ not reusable. No prior work has proved the conservativity of $F_{\leq}$ with equi-recursive types. This result is likely to be hard to prove because of the numerous non-modular changes in $F_{\leq}$ induced by the introduction of equi-recursive subtyping. Furthermore, in those works the full type systems are not provided. Abadi et al. [3] gave a sketch how to add iso-recursive subtyping to $F_{\leq}$ to encode objects. However, they did not prove any technical results about their extension. They translate the system to object calculi [1], in which the transitivity is built-in. In those papers on recursive types for $F_{\leq}$, they mostly focused on studying the subtyping relation of $F_{\leq}$, and the proofs were pen-and-paper rather than mechanized in a theorem prover.

There are many other extensions to $F_{\leq}$. Bounded existentials are also studied by Cardelli and Wegner [37]. Existential types can be encoded by universal types, thus we can obtain a form of bounded existentials for free in $F_{\leq}$ [37]. Another important extension is f-bounded quantification, firstly proposed by Canning et al. [29], then studied by Baldan, Ghelli, and Raffaeta [13] in terms of the basic theory. In f-bounded quantification, the bounded variables are allowed to appear in the bound. For example, a bound of the form $\forall(\alpha \leq F[\alpha]).\ B$, where F is a type-level function applied to α, is allowed in f-bounded quantification. We can encode polymorphic binary methods [25] and methods that have recursive types in their signatures with f-bounded quantification. For the example that we have showed in Chapter 8.1.2, the bound in the translate function is not Point but would have a form $F[\alpha] = \{x : \mathsf{Int},\ y : \mathsf{Int},\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$. Therefore, with f-bounded quantification the translate function could have the type:

$$\forall(\alpha \leq \{x : \mathsf{Int},\ y : \mathsf{Int},\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}).\ \alpha \to \alpha$$

Then the α can instantiate to Point or subtypes of Point, because $\mathsf{Point} \leq F[\mathsf{Point}]$. Note that

for subtyping statements such as $\alpha \leq \{x : \mathsf{Int},\ y : \mathsf{Int},\ \mathsf{move} : \mathsf{Int} \to \mathsf{Int} \to \alpha\}$ to hold, they must be interpreted using equi-recursive subtyping, since the f-bounds are normally records, and an iso-recursive type cannot be the subtype of a record. This approach is appealing because it can even deal with binary methods, where recursive types appear in negative positions. For example, with f-bounded quantification we can model bounds such as $\alpha \leq \{x : \mathsf{Int},\ \mathsf{eq} : \alpha \to \mathsf{Bool}\}$, and still have the expected subtyping relations.

Whereas we show that with the structural unfolding rule we can model positive cases of f-bounded quantification (such as translate) in $F^{\mu}_{\leq}$, we can only model a restrictive form of negative f-bounded quantification. For instance in $F^{\mu}_{\leq}$ we can have the bound $\alpha \leq \mu\ \mathsf{P}.\ \{x : \mathsf{Int},\ \mathsf{eq} : \mathsf{P} \to \mathsf{Bool}\}$ and we can instantiate α with P (where $P = \mu\ \mathsf{P}.\ \{x : \mathsf{Int},\ \mathsf{eq} : \mathsf{P} \to \mathsf{Bool}\}$). However, we would not be able to instantiate α with some types that have extra fields, such as μ P'. $\{x : \mathsf{Int},\ y : \mathsf{Int}, \mathsf{eq} : \mathsf{P'} \to \mathsf{Bool}\}$ for example. In contrast, f-bounded quantification allows such forms of instantiation. Nevertheless, given the overlap between some of the applications of iso-recursive types in $F^{\mu}_{\leq}$ and f-bounded quantification, we believe that it worthwhile to investigate whether f-bounded quantification can be avoided to deal with general binary methods.

9.2.2 Recursive records and existential types

Recursive records can encode objects [28, 49, 29]. Alternatively, existential types can also be used to encode objects [97], or they can be employed together with recursive types [27]. Pierce and Turner [97]'s object encoding is notable in that it requires only $F_{\leq}$, and does not employ recursive types. The *ORBE* encoding, presented by Abadi et al. [3] consists of recursive types, bounded existential quantification, records, and the structural unfolding rule. In their work, an interface is encoded as:

$$ORBE(I) \triangleq \mu\alpha.\ \exists(\beta \leq \alpha).\ \beta \times (\beta \to I(\beta))$$

As Bruce, Cardelli, and Pierce observe, the ORBE encoding requires full $F_{\leq}$ for the bounded quantification subtyping rule. When we try to compare two bounds, the type variable will be substituted with the existential types, which may result in bounds that are not equivalent. The overview paper by Bruce, Cardelli, and Pierce [28] makes a detailed comparison among different object encodings. Except for the *ORBE* encoding, $F^{\mu}_{\leq}$ could serve as a target for the existing object encodings. However, no complete formalization of $F_{\leq}$ with recursive types with desirable properties (such as type soundness and conservativity) existed at the time. Our work helps to further validate such encodings by providing a complete formalization of $F_{\leq}$ with recursive types, together with various desirable properties.

9.2.3 Algebraic datatypes and subtyping

Algebraic datatypes are a fundamental feature in modern functional programming languages, such as Haskell [71] and Ocaml [79]. However, such languages do not support subtyping

between datatypes. Hosoya et al. [73] discussed the interaction between mutually recursive datatypes and subtyping. They presented two variants of $F_{\leq}$ extending $F_{\leq}$ with user-defined datatype declarations. The first variant has user-defined subtyping declarations between datatypes, and can be viewed as having a form of nominal subtyping. The second variant allows structural subtyping among the datatypes. One advantage of employing user-defined datatypes is that it is simple to deal with formally, and that it allows mutually recursive datatype definitions easily. However, they do not support conventional recursive types of the form $\mu\alpha.\ A$ as we do in $F^{\mu}_{\leq}$. Moreover, they do not consider lower bounded quantification which, as argued in Chapter 8.1.5, is quite useful in a system targeting algebraic datatypes.

There has been some work integrating ML datatypes and OO classes [23, 87]. In the implementation of hierarchical extensible datatypes, methods are simulated via functions with dynamic dispatch. Those works are focused on the design of intermediate languages that have complex constructs such as classes or datatypes. In contrast, we develop foundational calculi, where more complex constructs can be encoded.

Finally, Poll [101] investigated the categorical semantics of datatypes with subtyping and a limited form of inheritance on datatypes, improving our understanding on the relation between categorical datatypes and object types.

Oliveira [90] showed encodings of algebraic datatypes with subtyping assuming a variant of $F_{\leq}$ extended with records, recursive types and higher kinds. He showed that adding subtyping to datatypes allows solving the Expression Problem [116]. However, as we mentioned in Chapter 8.1.3, he did not formalize the $F_{\leq}$ extension, although he showed a translation of the encoding into Scala. Moreover, his encoding is more complex than ours because he employs upper bounded quantification with higher kinds. In Chapter 8.1.5, we showed that first-order lower bounded quantification in $F^{\mu}_{\leq\geq}$, together with the structural folding rule enables such encodings. Like for encodings of objects, our work is helpful to further validate such encodings formally.

9.2.4 Disjoint intersection types and record calculi

Disjoint intersection types were originally proposed by Oliveira et al. [89]. Such calculi have intersection types as well as a merge operator [106, 57] with a disjointness restriction to ensure the determinism of the language. Follow-up work [4, 20, 19] provides more advanced features, such as disjoint polymorphism and distributive subtyping and first-class trait, built upon the original work. With all these features together, an alternative paradigm called *Compositional Programming* (CP) is proposed [21, 119]. CP allows for a very modular programming style where the Expression Problem [116] can be solved naturally. A limitation of existing calculi with disjoint intersection types is that they do not support recursive types, which are important to encode binary methods [25] or, more generally, recursive object types. The λ_i^{μ} calculus addresses this limitation and shows, for the first time, a calculus with disjoint intersection types and recursive types.

The merge operator generalizes concatenation by allowing values of any types (not just record types) to be merged. As we described in the introduction the interaction between subtyping and record concatenation is quite tricky. Cardelli and Mitchell [36] observed the problem [36], but did not provide a solution. Instead, they decided to use record extension and restriction operators instead of concatenation. One solution adopted by some calculi [105, 102, 92, 69] is to distinguish between records that can be concatenated, and records that have subtyping. The choice is mutually exclusive: records that can be concatenated cannot have subtyping and vice-versa. Such an approach would prevent Cardelli and Mitchell's `f2` example in Chapter 1.1.3 from type-checking. Calculi with disjoint intersection types, including λ_i^{μ}, offer a different solution by adopting a type-directed semantics, which ensures that fields hidden by subtyping are also hidden at runtime. This allows concatenation and subtyping to be used together.

As far as we know, no full formalization of a calculus supports subtyping, record concatenation and recursive subtyping at the same time. In Cardelli's $F_{<:_\rho}$ calculus [33] equi-recursive subtyping is assumed to be an extension to record subtyping and record concatenation but no proofs were provided. Palsberg and Zhao's work [92] shows supporting subtyping, record concatenation and recursive types (but no recursive subtyping) together for type inference is NP-complete.

9.2.5 Dependent object types

Dependent object types (DOT), the foundation of *Scala*, also considers intersection types, a special form of equi-recursive type, and a generalized form of bounded quantification [6, 108]. The subtyping rules for intersection types are similar to our λ_i^{μ}. The generalized form of bounded quantification subsumes both upper and lower bounded quantification, which are present in $F_{\leq}^{\mu}$ and $F_{\leq\geq}^{\mu}$. With conventional recursive types $\mu\alpha.\ A$, α stands for the recursive type itself. In DOT, the recursive type is of the form $\mu\ this.\ A$, where *this* is the (run-time) self-reference. This construct, in combination with the form of dependent types supported in DOT allows for interesting applications that cannot be modelled with conventional recursive types. However, there are no records and record concatenation, since objects are directly modelled rather than being encoded via records. Moreover, the rules for recursive types employed in some variants of DOT [108, 117, 66] are mostly structural and employ an inductive definition of subtyping.

The key subtyping rules in the DOT variant by Rompf and Amin [108] are shown next:

$$\frac{\Gamma, z : T_1^z \vdash T_1^z <: T_2^z}{\Gamma \vdash \mu z.\ T_1^z <: \mu z.\ T_2^z}\ \text{BindX} \qquad \frac{\Gamma, z : T_1^z \vdash T_1^z <: T_2}{\Gamma \vdash \mu z.\ T_1^z <: T_2}\ \text{BindI}$$

$$\frac{\Gamma \vdash x : T^x}{\Gamma \vdash x : \mu z.\ T^z}\ \text{VarPack} \qquad \frac{\Gamma \vdash x : \mu z.\ T^z}{\Gamma \vdash x : T^x}\ \text{VarUnpack}$$

We adapt the notation employed by Rompf and Amin [108] for recursive types to our notation. In the rules, T^z denotes that variable z is free in type T. Rule BindX is in essence a one-step finite unfolding of the recursive type, leading to an inductive definition of subtyping. Thus, such definition is unlike approaches to equi-recursive subtyping, where a non-structural coinductive formulation of subtyping is used instead. The second rule for recursive types (rule BindI) is a special case where a recursive type $\mu z.\ T_1^z$ is a subtype of another type T_2 if T_2 does not contain the recursive variable z. A difference between recursive types $\mu z.\ T^z$ in DOT and the ones in this thesis is that in DOT z is a *term* variable instead of a type variable. In DOT, recursive types can be used in combination with *path-dependent types* [9], to denote types such as $z.L$, where z represents a (possibly recursive) term with a type member L. Because of this design, the typing rules that introduce and eliminate recursive types [108], are defined on variables. Unlike our formulation of subtyping, which is algorithmic, DOT's formulation of subtyping is usually presented in a declarative form. Undecidability is an important problem with DOT's formulation of subtyping [74], and the existing decidable fragments of DOT lack transitivity [74, 85]. The research on DOT has been intimately related to $F_{\leq}$. For instance, Amin and Rompf [8] explain many of the features of DOT by incrementally extending $F_{\leq}$. In addition, there have been various attempts to prove the undecidability of DOT by a reduction to the undecidability problem in $F_{\leq}$. Although, as Hu and Lhoták [74] observed, DOT is not conservative over $F_{\leq}$. Thus an undecidability result for DOT cannot be proved by reduction to the undecidability of full $F_{\leq}$. While $F^{\mu}_{\leq\geq}$ does not have all the features of DOT, our results can potentially help in research in that area, where the decidable fragments of DOT lack important properties such as transitivity. In addition $F^{\mu}_{\leq\geq}$ preserves the conservativity over $F_{\leq}$, while DOT does not.

Chapter 10

Conclusion and Future Work

In this thesis, we revisit the problem of iso-recursive subtyping and come up with novel declarative and algorithmic formulations of subtyping. We pay special attention to the metatheory, which is fully formalized in the Coq theorem prover. We believe that our work significantly improves the understanding of iso-recursive subtyping, and provides a platform for further developments in this area. More practically, the double unfolding rule and nominal unfolding rule are easy to integrate in existing calculi and this work presents the proof techniques needed to prove standard properties (such as transitivity and type soundness). Moreover, we show that it is easy to employ our algorithmic subtyping rules to subtyping relations that are not antisymmetric. For example, we study the combination of iso-recursive subtyping and intersection types, and showed that this combination is well-behaved using the novel nominal unfolding rules. While both the double unfoldings and the nominal unfoldings are equivalent in simpler settings, they are different in the presence of more advanced type system features. We have found a counter-example for intersection types, and we suspect that this is also the case for bounded quantification. Fortunately, the nominal unfolding rules seem to work in all settings so far. From a practical point of view, we also discussed how to employ iso-recursive subtyping in object encodings. As a concrete example, our λ_i^μ calculus illustrates that the 3 features, recursive types, extensible record types and intersection types, can be put together in a single calculus. Object types can be modelled with recursive records and multiple inheritance can be modelled via intersection types and record concatenation. Therefore our λ_i^μ calculus can be used to provide simple encodings for objects. Finally, our $F_{\leq}^\mu$ calculus illustrates how to integrate iso-recursive types and kernel $F_{\leq}$. We obtain a transitive and decidable subtyping relation, while the full calculus is shown to be conservative over $F_{\leq}$ and is proven to be type-sound. $F_{\leq}^\mu$ and $F_{\leq\geq}^\mu$ could serve as the theoretic foundation for object encodings and encodings of algebraic datatypes with subtyping. With the renewed interest on recursive types and bounded quantification due to the DOT calculus, we believe that our work is also helpful to find calculi with most features in DOT, while retaining desirable properties, such as decidability and transitivity of subtyping.

There are a few interesting directions for future work.

10.1 Recursive Subtyping with One-Step Unfolding

As we describe at Chapter 3, currently, in our subtyping rules for iso-recursive types, the number of times that the left and the right types are unfolded is required to be exactly the same. The cases where we would unfold the recursive types a different number of types on the left and on the right, are not allowed. However, as Ligatti et al. [84] point out, allowing subtyping between an iso-recursive type and its one-step unfolding does not break the type soundness. The example they gave is:

$$\mu i.\{\text{sub} : (\mu i'.\{\text{sub} : i' \to \text{unit}\}) \to \text{unit}, \text{min} : \text{unit} \to \text{int}\} \leq \mu i.\{\text{sub} : i \to \text{unit}\}$$

the left type can be regarded as the one-step unfolding of the right type with some extra fields. Our subtyping formulation for iso-recursive types is not able to capture this subtyping relationship. However, we can add two extra rules:

$$\frac{\Gamma \vdash [\alpha \mapsto A^{\alpha}]\, A \leq [\alpha \mapsto B^{\alpha}]\, B}{\Gamma \vdash A^{\alpha} \leq \mu\alpha.\, B}\ \text{S-COMPLEFT} \qquad \frac{\Gamma \vdash [\alpha \mapsto A^{\alpha}]\, A \leq [\alpha \mapsto B^{\alpha}]\, B}{\Gamma \vdash \mu\alpha.\, A \leq B^{\alpha}}\ \text{S-COMPRIGHT}$$

The key point of these two rules rule S-COMPLEFT and rule S-COMPRIGHT is reusing the label/name. Note that in our Coq formalization, the α in $\mu\alpha.\, A$ and $\mu\alpha.\, B$ is practically a bounded variable, and during the derivation it will be converted to a free variable α, which is already stored in the context. With those two new rules, we can derive the example above. For saving space, let us denote $\mu i'.\{\text{sub} : i' \to \text{unit}\}$ as A and $\{\text{sub} : i \to \text{unit}\}$ as B.

$$\dfrac{\dfrac{\dfrac{\dfrac{i \vdash \{\text{sub} : B^i \to \text{unit}\} \leq \{\text{sub} : B^i \to \text{unit}\}}{i \vdash \{\text{sub} : i \to \text{unit}\}^i \leq \mu i'.\{\text{sub} : i' \to \text{unit}\}} \qquad \dfrac{i \vdash B^i \leq A \qquad i \vdash \text{unit} \leq \text{unit}}{i \vdash A \to \text{unit} \leq B^i \to \text{unit}}}{i \vdash \{\text{sub} : A \to \text{unit}, \text{min} : \text{unit} \to \text{int}\} \leq \{\text{sub} : B^i \to \text{unit}\}}}{\vdash \mu i.\{\text{sub} : A \to \text{unit}, \text{min} : \text{unit} \to \text{int}\} \leq \mu i.\{\text{sub} : i \to \text{unit}\}}$$

From the derivation tree, we can see that once we compare a label type to a recursive type in rule S-COMPLEFT and rule S-COMPRIGHT, it actually triggers the unfolded recursive type to unfold one more time. Therefore, during the derivation, $\mu i.\{\text{sub} : A \to \text{unit}, \text{min} : \text{unit} \to \text{int}\}$ unfolds one time and $\mu i.\{\text{sub} : i \to \text{unit}\}$ unfolds two times, and then they can compare directly with the same number of unfoldings.

We should note that, although adding two new rules to our formulations presented in this thesis does make them more powerful, they are not complete. In Chapter 9.1.2, we give a counter-example that shows that $\mu\alpha.\, \alpha$ is subtype of $\mu\alpha.(\mu\beta.\, \alpha)$ under the rules by Ligatti et al. [84]. However, under our formulation, even when we add two new rules, they are still not subtype.

10.2 Distributive Iso-Recursive Subtyping

In Chapter 7, our λ_i^μ type system combines the features of recursive types and intersection types. When a system involves intersection types, it is quite useful to have distributive law for subtyping [17].

$$\text{S-DISTARR}\quad \frac{}{\Gamma \vdash (A_1 \to A_2)\&(A_1 \to A_3) \leq A_1 \to A_2\&A_3}$$

For example, rule S-DISTARR is the distributive law for subtyping involves intersection types and function types. By adding this rule, we can claim that $(A_1 \to A_2)\&(A_1 \to A_3)$ and $A_1 \to (A_2\&A_3)$ are equivalent via:

$$\frac{\dfrac{\Gamma \vdash A_1 \leq A_1 \qquad \Gamma \vdash A_2\&A_3 \leq A_2}{\Gamma \vdash A_1 \to A_2\&A_3 \leq A_1 \to A_2} \qquad \dfrac{\Gamma \vdash A_1 \leq A_1 \qquad \Gamma \vdash A_2\&A_3 \leq A_3}{\Gamma \vdash A_1 \to A_2\&A_3 \leq A_1 \to A_3}}{\Gamma \vdash A_1 \to A_2\&A_3 \leq (A_1 \to A_2)\&(A_1 \to A_3)}$$

It is not hard to support the distributive law for subtyping involving intersection types and record types or universal types. However, if we want to support the distributive law for subtyping involving intersection types and recursive types, we will lose the equivalence property. Concretely, we might want to add the following rule:

$$\text{S-DISTREC}\quad \frac{}{\Gamma \vdash (\mu\alpha.\ A_1)\&(\mu\alpha.\ A_2) \leq \mu\alpha.\ A_1\&A_2}$$

When the recursive variable occurs positively, $(\mu\alpha.\ A_1)\&(\mu\alpha.\ A_2)$ and $\mu\alpha.\ A_1\&A_2$ are equivalent, while when the recursive variable occurs negatively, the equivalence is not satisfied. Let $A = (\alpha \to \text{nat})\&(\alpha \to \text{char})$, a counter-example is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{Derivation fails here}}{\alpha \vdash \alpha \to \text{char} \leq \alpha \to \text{nat}} \quad \cdots}{\alpha \vdash \alpha \to \text{char} \leq (\alpha \to \text{nat})\&(\alpha \to \text{char})} \qquad \dfrac{\alpha \vdash (\alpha \to \text{char})^\alpha \leq A^\alpha \qquad \cdots}{\alpha \vdash A^\alpha \to \text{char} \leq (\alpha \to \text{char})^\alpha \to \text{char}}}{\alpha \vdash (A^\alpha \to \text{nat})\&(A^\alpha \to \text{char}) \leq (\alpha \to \text{char})^\alpha \to \text{char}}}{\vdash \mu\alpha.\ (\alpha \to \text{nat})\&(\alpha \to \text{char}) \leq \mu\alpha.\ \alpha \to \text{char}} \quad \cdots}{\vdash \mu\alpha.\ (\alpha \to \text{nat})\&(\alpha \to \text{char}) \leq (\mu\alpha.\ \alpha \to \text{nat})\&(\mu\alpha.\ \alpha \to \text{char})}$$

Unfortunately, many approaches [20, 95, 75] somehow rely on the equivalence property, thus we have no idea about how to add rule S-DISTREC to our λ_i^μ type system yet. If we can support the distributive law for recursive subtyping, we can encode nested composable traits with binary methods [119].

10.3 Full $F_{\leq}$ with Iso-Recursive Types

Our $F_{\leq}^{\mu}$ calculus is built upon the kernel $F_{\leq}$ via integrating rule S-KERNELALL and rule S-FREC. As we discussed in Chapter 9.2.2, if we want to model the *ORBE* encoding [3], the combination of iso-recursive subtyping and kernel $F_{\leq}$ is not enough. For encoding imperative object types,

investigating extensions of rule S-FREC with rule S-FULLALL instead of rule S-KERNELALL is a clear avenue for future work.

Unfortunately, we cannot apply the technique we used in Chapter 8.3.1 to full $F_{\leq}$. In the proof of generalized unfolding lemma for $F^{\mu}_{\leq}$, we pick up an intermediate type between the two bounds, and the bounds are required to be equivalent when they are compared, thus the intermediate type is also equivalent to the bounds. However, in full $F_{\leq}$, in the context, the type C and D will swap due to the contravariance, thus the intermediate type is not equivalent to the bounds, and the proof breaks.

We believe that it is feasible to combine iso-recursive subtyping with full $F_{\leq}$. The current approach in Chapter 8.3.1 for generalized unfolding lemma does not try to generalize the context, in which we think it is the key point for rescue the proof, so that we can process to the type soundness. Other properties, such as reflexivity and transitivity, like $F^{\mu}_{\leq}$, are easy to prove. We note that, since full $F_{\leq}$ is not decidable [98], the combination of iso-recursive subtyping and full $F_{\leq}$ is obviously not decidable neither.

10.4 Getting F-Bounded Polymorphism for Free

In Chapter 10.1, we discuss a possible way to enhance the expressive power of our subtyping rules for iso-recursive types in a simpler type system. If we apply this approach to the more complete type system, in order words, extending $F^{\mu}_{\leq}$ with those two new rules rule S-COMPLEFT and rule S-COMPRIGHT, we are able to model f-bounded polymorphism [29].

In Chapter 8.1.2, we showed that the usage of structural unfolding rule enabled us to model positive f-bounded polymorphism. However, for the bound where the variable appears negatively, such as $\alpha \leq \mu\mathsf{P}.\ \{\mathsf{eq} : \mathsf{P} \to \mathsf{bool}\}$, we can only instantiate α with the recursive type itself, disallowing having the extra fields.

The addition of rule S-COMPLEFT and rule S-COMPRIGHT solves this problem. For example, we can have the following subtyping statement:

$$\mu\mathsf{P}.\ \{\mathsf{eq} : (\mu\mathsf{P'}.\ \{\mathsf{eq} : \mathsf{P'} \to \mathsf{bool}, \mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}\}) \to \mathsf{bool}, \mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}, \mathsf{color} : \mathsf{String}\}$$
$$\leq$$
$$\mu\mathsf{P}.\ \{\mathsf{eq} : \mathsf{P} \to \mathsf{bool}, \mathsf{x} : \mathsf{Int}, \mathsf{y} : \mathsf{Int}\}$$

in which the left type represent an encoding of the colored point, while the right type (denoted as Point) is the encoding of the base point. Then the base class of point is defined as:

```
class Point {
  eq: function (p: Point): bool = {
    p.x == this.x && p.y == this.y
  }
  x: Int;
  y: Int;
}
```

and the class colored point is defined as:

```
class ColoredPoint extends Point {
  color: String;
}
```

where the common fields are inherited from the base class, and the polymorphic binary method distance is defined as:

```
function distance(P <: Point, p1: P, p2: P): Int = sqrt(sqr(p1.x
  - p2.x) + sqr(p1.y - p2.y))
```

10.5 Higher-Order Type System with Iso-Recursive Types

We already discussed the addition of iso-recursive subtyping in both first-order type system (simply typed lambda calculus, STLC) in Chapter 3 and 4, and second-order type system (simply typed lambda calculus with polymorphism, System F) in Chapter 8. It is also interesting to see how iso-recursive subtyping can be integrated into a higher-order type system. In STLC, a value is dependent on a type. In System F, a value is dependent on a type and a term. In a higher-order type system, a type can abstract over another type, which enables the type constructors. Some more complex features, such as abstract data structures and general self types, can be modeled. The first higher-order type system, F_ω [68], does not have subtyping. Later, F_ω is extended to $F_{\omega\leq}$ [34, 47, 94], where the subtyping is added. For targeting a higher-order type system with recursive types and subtyping, Bruce and Mitchell [26] extended $F_{\omega\leq}$ to $F^{\mu}_{\omega\leq}$, which is also used to interpret f-bounded polymorphism [29]. Note that the recursive types in Bruce and Mitchell [26]'s work are interpreted as equi-recursive types, which is significantly more complex than what we proposed in Chapter 10.4. A simpler calculus of $F^{\mu}_{\omega\leq}$, by replacing the equi-recursive types with iso-recursive types, is worthy to explore.

Bibliography

[1] M. Abadi and L. Cardelli. *A theory of objects*. Springer Science & Business Media, 2012.

[2] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. "Explicit Substitutions". In: *J. Funct. Program.* 1.4 (1991), pp. 375–416.

[3] M. Abadi, L. Cardelli, and R. Viswanathan. "An interpretation of objects and object types". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1996, pp. 396–409.

[4] J. Alpuim, B. C. d. S. Oliveira, and Z. Shi. "Disjoint polymorphism". In: *European Symposium on Programming*. Springer. 2017, pp. 1–28.

[5] R. M. Amadio and L. Cardelli. "Subtyping recursive types". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15.4 (1993), pp. 575–631.

[6] N. Amin, S. Grütter, M. Odersky, T. Rompf, and S. Stucki. "The essence of dependent object types". In: *A List of Successes That Can Change the World*. Springer, 2016, pp. 249–272.

[7] N. Amin, A. Moors, and M. Odersky. "Dependent object types". In: *19th International Workshop on Foundations of Object-Oriented Languages*. CONF. 2012.

[8] N. Amin and T. Rompf. "Type soundness proofs with definitional interpreters". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017, pp. 666–679.

[9] N. Amin, T. Rompf, and M. Odersky. "Foundations of Path-Dependent Types". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA '14. Association for Computing Machinery, 2014.

[10] A. W. Appel and A. P. Felty. "A semantic model of types and machine instructions for proof-carrying code". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2000, pp. 243–253.

[11] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. "Engineering formal metatheory". In: *Acm sigplan notices* 43.1 (2008), pp. 3–15.

[12] M. Backes, C. Hriţcu, and M. Maffei. "Union, intersection and refinement types and reasoning about type disjointness for secure protocol implementations". In: *Journal of Computer Security* 22.2 (2014), pp. 301–353.

[13] P. Baldan, G. Ghelli, and A. Raffaeta. "Basic theory of F-bounded quantification". In: *Information and Computation* 153.2 (1999), pp. 173–237.

[14] F. Barbanera and M. Dezani-Ciancaglini. "Intersection and union types". In: *International Symposium on Theoretical Aspects of Computer Software*. Springer. 1991, pp. 651–674.

[15] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. "Intersection and Union Types: Syntax and Semantics". In: *Information and Computation* 119.2 (June 1995), pp. 202–230.

[16] F. Barbanera, M. Dezaniciancaglini, and U. Deliguoro. "Intersection and union types: syntax and semantics". In: *Information and Computation* 119.2 (1995), pp. 202–230.

[17] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. "A filter lambda model and the completeness of type assignment". In: *The journal of symbolic logic* 48.4 (1983), pp. 931–940.

[18] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. "Refinement types for secure implementations". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33.2 (2011), pp. 1–45.

[19] X. Bi and B. C. d. S. Oliveira. "Typed first-class traits". In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. 2018.

[20] X. Bi, B. C. d. S. Oliveira, and T. Schrijvers. "The essence of nested composition". In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. 2018.

[21] X. Bi, N. Xie, B. C. d. S. Oliveira, and T. Schrijvers. "Distributive disjoint polymorphism for compositional programming". In: *European Symposium on Programming*. Springer, Cham. 2019, pp. 381–409.

[22] C. Böhm and A. Berarducci. "Automatic synthesis of typed Lambda-programs on term algebras". In: *Theoretical Computer Science* 39.2-3 (1985).

[23] F. Bourdoncle and S. Merz. "Type checking higher-order polymorphic multi-methods". In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1997, pp. 302–315.

[24] M. Brandt and F. Henglein. "Coinductive axiomatization of recursive type equality and subtyping". In: vol. 1210. Full version in *Fundamenta Informaticae*, 33:309–338, 1998. Apr. 1997, pp. 63–81.

[25] K. Bruce, L. Cardeli, G. Castagna, T. H. O. Group, G. T. Leavens, and B. Pierce. "On binary methods". In: *Theory and Practice of Object Systems* 1.3 (1996).

[26] K. Bruce and J. C. Mitchell. "PER models of subtyping, recursive types and higher-order polymorphism". In: *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1992, pp. 316–327.

[27] K. B. Bruce. "A paradigmatic object-oriented programming language: Design, static typing and semantics". In: *Journal of Functional Programming* 4.2 (1994), pp. 127–206.

[28] K. B. Bruce, L. Cardelli, and B. C. Pierce. "Comparing Object Encodings". In: vol. 155. 1. 1999.

[29] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. "F-Bounded Polymorphism for Object-Oriented Programming". In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA 1989. Imperial College, London, United Kingdom, 1989.

[30] L. Cardelli. "A Language with Distributed Scope". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. 1995, pp. 286–297.

[31] L. Cardelli. "A semantics of multiple inheritance". In: *Semantics of Data Types*. 1984.

[32] L. Cardelli. "Amber". In: *LITP Spring School on Theoretical Computer Science*. Springer. 1985, pp. 21–47.

[33] L. Cardelli. *Extensible records in a pure calculus of subtyping*. Digital. Systems Research Center, 1992.

[34] L. Cardelli. "Structural subtyping and the notion of power type". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1988, pp. 70–79.

[35] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. "An extension of system F with subtyping". In: *Information and computation* 109.1-2 (1994), pp. 4–56.

[36] L. Cardelli and J. C. Mitchell. "Operations on records". In: *Mathematical Structures in Computer Science* 1.1 (1991), pp. 3–48.

[37] L. Cardelli and P. Wegner. "On understanding types, data abstraction, and polymorphism". In: *ACM Computing Surveys (CSUR)* 17.4 (1985), pp. 471–523.

[38] F. Cardone. "Recursive types for Fun". In: *Theoretical Computer Science* 83.1 (1991), pp. 39–56.

[39] G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. "Foundations of session types". In: *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming*. 2009, pp. 219–230.

[40] G. Castagna and A. Frisch. "A Gentle Introduction to Semantic Subtyping". In: *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '05. 2005.

[41] G. Castagna and B. C. Pierce. "Decidable Bounded Quantification". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '94. Portland, Oregon, USA: Association for Computing Machinery, 1994, pp. 151–162. ISBN: 0897916360.

[42] A. Charguéraud. "The Locally Nameless Representation". In: *Journal of Automated Reasoning* (2011). 10.1007/s10817-011-9225-2, pp. 1–46. ISSN: 0168-7433.

[43] T.-C. Chen, M. Dezani-Ciancaglini, and N. Yoshida. "On the preciseness of subtyping in session types". In: *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. 2014, pp. 135–146.

[44] R. Chugh. "IsoLATE: A type system for self-recursion". In: *European Symposium on Programming*. Springer. 2015, pp. 257–282.

[45] A. Church. "A set of postulates for the foundation of logic". In: *Annals of Mathematics* 33.2 (1932), pp. 346–366.

[46] D. Colazzo and G. Ghelli. "Subtyping recursion and parametric polymorphism in kernel fun". In: *Information and Computation* 198.2 (2005), pp. 71–147.

[47] A. B. Compagnoni. *Higher-order subtyping with intersection types*. [Sl: sn], 1995.

[48] A. B. Compagnoni and B. C. Pierce. "Higher-order intersection types and multiple inheritance". In: *Mathematical Structures in Computer Science* 6.5 (1996), pp. 469–501.

[49] W. R. Cook, W. Hill, and P. S. Canning. "Inheritance is Not Subtyping". In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '90. 1989, pp. 125–135.

[50] W. R. Cook and J. Palsberg. "A denotational semantics of inheritance and its correctness". In: *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*. 1989.

[51] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. "Functional Characters of Solvable Terms". In: *Mathematical Logic Quarterly* 27.2-6 (1981), pp. 45–58.

[52] K. Crary, R. Harper, and S. Puri. "What is a Recursive Module?" In: *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*. PLDI '99. 1999.

[53] P.-L. Curien and G. Ghelli. "Coherence of subsumption, minimum typing and typechecking in $F_{\leq}$". In: *Mathematical structures in computer science* 2.1 (1992), pp. 55–91.

[54] F. M. Damm. "Subtyping with union types, intersection types and recursive types". In: *International Symposium on Theoretical Aspects of Computer Software*. Springer. 1994, pp. 687–706.

[55] N. A. Danielsson and T. Altenkirch. "Subtyping, declaratively". In: *International Conference on Mathematics of Program Construction*. Springer. 2010, pp. 100–118.

[56] D. Duggan. "Type-safe linking with recursive DLLs and shared libraries". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 24.6 (2002), pp. 711–804.

[57] J. Dunfield. "Elaborating intersection and union types". In: *Journal of Functional Programming* 24.2-3 (2014), pp. 133–165.

[58] Eclipse Foundation. *Ceylon*. Version 1.3.3. 2017. URL: https://ceylon-lang.org/.

[59] EPFL. *Scala 3*. 2021. URL: https://www.scala-lang.org/.

[60] Facebook. *Flow*. Version 0.154.0. 2021. URL: https://flow.org/.

[61] A. Frisch, G. Castagna, and V. Benzaken. "Semantic subtyping". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, pp. 137–146.

[62] V. Gapeyev, M. Levin, and B. Pierce. "Recursive Subtyping Revealed". In: *Journal of Functional Programming* 12.6 (2003). Preliminary version in *International Conference on Functional Programming (ICFP)*, 2000. Also appears as Chapter 21 of *Types and Programming Languages* by Benjamin C. Pierce (MIT Press, 2002)., pp. 511–548.

[63] S. Gay and M. Hole. "Subtyping for session types in the pi calculus". In: *Acta Informatica* 42.2-3 (2005), pp. 191–225.

[64] S. J. Gay and V. T. Vasconcelos. "Linear type theory for asynchronous session types". In: *Journal of Functional Programming* 20.1 (2010), pp. 19–50.

[65] G. Ghelli. "Recursive types are not conservative over $F_{\leq}$". In: *International Conference on Typed Lambda calculi and Applications*. Springer. 1993, pp. 146–162.

[66] P. G. Giarrusso, L. Stefanesco, A. Timany, L. Birkedal, and R. Krebbers. "Scala Step-by-Step: Soundness for DOT with Step-Indexed Logical Relations in Iris". In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020).

[67] J.-Y. Girard. "Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur". Thèse d'état. Université de Paris 7, 1972.

[68] J.-Y. Girard. "Une extension de L'interpretation de gödel a L'analyse, et son application a L'elimination des coupures dans L'analyse et la theorie des types". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 63. Elsevier, 1971, pp. 63–92.

[69] N. Glew. "An Efficient Class and Object Encoding". In: *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '00. Association for Computing Machinery, 2000, pp. 311–324.

[70] B. Greenman, F. Muehlboeck, and R. Tate. "Getting F-bounded polymorphism into shape". In: *ACM SIGPLAN Notices* 49.6 (2014), pp. 89–99.

[71] Haskell Development Team. *Haskell*. 1990. URL: https://www.haskell.org/.

[72] M. Hofmann and B. C. Pierce. "Positive subtyping". In: *Information and Computation* 126.1 (1996), pp. 11–33.

[73] H. Hosoya, B. C. Pierce, D. N. Turner, et al. "Datatypes and subtyping". In: *Unpublished manuscript. Available http://www. cis. upenn. edu/~ bcpierce/papers/index. html* (1998).

[74] J. Z. Hu and O. Lhoták. "Undecidability of dsub and its decidable fragments". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–30.

[75] X. Huang and B. C. d. S. Oliveira. "Distributing intersection and union types with splits and duality (functional pearl)". In: *Proceedings of the ACM on Programming Languages* 5.ICFP (2021), pp. 1–24.

[76] X. Huang and B. C. d. S. Oliveira. "A Type-Directed Operational Semantics For a Calculus with a Merge Operator". In: *34th European Conference on Object-Oriented Programming (ECOOP 2020)*. Vol. 166. 2020.

[77] X. Huang, J. Zhao, and B. C. d. S. Oliveira. "Taming the merge operator: a type-directed operational semantics approach". In: *Journal of Functional Programming* (2021).

[78] A. Igarashi, B. C. Pierce, and P. Wadler. "Featherweight Java: a minimal core calculus for Java and GJ". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.3 (2001), pp. 396–450.

[79] INRIA. *OCaml*. 1987. URL: https://ocaml.org/.

[80] A. Jeffrey. "A symbolic labelled transition system for coinductive subtyping of F^< types". In: *2001 IEEE Conference on Logic and Computer Science (LICS 2001)*. 2001, p. 323.

[81] A. J. Kennedy and B. C. Pierce. "On decidability of nominal subtyping with variance". In: (2006).

[82] L. Zhou, Y. Zhou, and B. C. d. S. Oliveira. "Recursive Subtyping for All". In: *57th Symposium on Principles of Programming Languages (POPL 2023) (Conditional accepted)*. 2023.

[83] J. Lee, J. Aldrich, T. Shaw, and A. Potanin. "A Theory of Tagged Objects". In: *European Conference on Object-Oriented Programming (ECOOP)*. 2015.

[84] J. Ligatti, J. Blackburn, and M. Nachtigal. "On subtyping-relation completeness, with an application to iso-recursive types". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39.1 (2017), pp. 1–36.

[85] J. Mackay, A. Potanin, J. Aldrich, and L. Groves. "Decidable subtyping for path dependent types". In: *Proceedings of the ACM on Programming Languages* 4.POPL (2019), pp. 1–27.

[86] Microsoft. *TypeScript*. Version 4.3. 2021. URL: https://www.typescriptlang.org/.

[87] T. Millstein, C. Bleckner, and C. Chambers. "Modular typechecking for hierarchically extensible datatypes and functions". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 26.5 (2004), pp. 836–889.

[88] J. H. J. Morris. "Lambda-calculus models of programming languages." PhD thesis. Massachusetts Institute of Technology, 1969.

[89] B. C. d. S. Oliveira, Z. Shi, and J. Alpuim. "Disjoint intersection types". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. 2016, pp. 364–377.

[90] B. C. d. S. Oliveira. "Modular visitor components". In: *European Conference on Object-Oriented Programming*. Springer. 2009, pp. 269–293.

[91] B. C. d. S. Oliveira, S. Cui, and B. Rehman. "The Duality of Subtyping". In: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)*. Ed. by R. Hirschfeld and T. Pape. LIPIcs. 2020.

[92] J. Palsberg and T. Zhao. "Type Inference for Record Concatenation and Subtyping". In: *Inf. Comput.* 189.1 (Feb. 2004).

[93] M. Parigot. "Recursive programming with proofs". In: *Theoretical Computer Science* 94.2 (1992), pp. 335–356.

[94] B. Pierce and M. Steffen. "Higher-order subtyping". In: *Theoretical computer science* 176.1-2 (1997), pp. 235–282.

[95] B. C. Pierce. *A decision procedure for the subtype relation on intersection types with bounded variables*. Carnegie-Mellon University. Department of Computer Science, 1989.

[96] B. C. Pierce. *Bounded quantification with bottom*. Tech. rep. Citeseer, 1997.

[97] B. C. Pierce and D. N. Turner. "Simple type-theoretic foundations for object-oriented programming". In: *Journal of functional programming* 4.2 (1994), pp. 207–247.

[98] B. C. Pierce. "Bounded quantification is undecidable". In: *Information and Computation* 112.1 (1994), pp. 131–165.

[99] B. C. Pierce. "Intersection types and bounded polymorphism". In: *Mathematical Structures in Computer Science* 7.2 (1997), pp. 129–193.

[100] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[101] E. Poll. "Subtyping and Inheritance for Categorical Datatypes: Preliminary Report (Type Theory and its Applications to Computer Systems)". In: *Kyoto University Research Information Repository* 1023 (1998), pp. 112–125.

[102] F. Pottier. "A 3-part type inference engine". In: *European Symposium on Programming*. Springer. 2000, pp. 320–335.

[103] F. Pottier. "Syntactic soundness proof of a type-and-capability system with hidden state". In: *Journal of functional programming* 23.1 (2013), pp. 38–144.

[104] G. Pottinger. "A type assignment for the strongly normalizable λ-terms". In: *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), pp. 561–577.

[105] D. Rémy. "A case study of typechecking with constrained types: Typing record concatenation". Presented at the workshop on Advances in types for computer science at the Newton Institute, Cambridge, UK. 1995.

[106] J. C. Reynolds. "Preliminary design of the programming language Forsythe". In: (1988).

[107] J. C. Reynolds. "Towards a theory of type structure". In: *Colloque sur la Programmation*. Springer. 1974, pp. 408–425.

[108] T. Rompf and N. Amin. "Type soundness for dependent object types (DOT)". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2016, pp. 624–641.

[109] D. Sangiorgi and R. Milner. "The Problem of "Weak Bisimulation up to"". In: *CONCUR*. Vol. 630. 1992, pp. 32–46.

[110] D. Scott. "A system of functional abstraction". Lectures delivered at University of California, Berkeley, California, USA, 1962/63. 1962.

[111] J. G. Siek and S. Tobin-Hochstadt. "The recursive union of some gradual types". In: *A List of Successes That Can Change the World*. Springer, 2016, pp. 388–410.

[112] M. Solomon. "Type definitions with parameters". In: *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 1978, pp. 31–38.

[113] C. Stone and R. Harper. *A Type-Theoretic Account of Standard ML 1996*. Tech. rep. CMU-CS-96-136. School of Computer Science, Carnegie Mellon University, May 1996.

[114] N. Swamy, J. Chen, C. Fournet, P.-Y. Strub, K. Bhargavan, and J. Yang. "Secure Distributed Programming with Value-Dependent Types". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2011. 2011.

[115] J. C. Vanderwaart, D. Dreyer, L. Petersen, K. Crary, R. Harper, and P. Cheng. "Typed Compilation of Recursive Datatypes". In: *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*. 2003, pp. 98–108.

[116] P. Wadler. "The Expression Problem". discussion on the Java Genericity mailing list. 1998.

[117] F. Wang and T. Rompf. "Towards strong normalization for dependent object types (DOT)". In: *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[118] Y. Yang and B. C. d. S. Oliveira. "Pure iso-type systems". In: *Journal of Functional Programming* 29 (2019).

[119] W. Zhang, Y. Sun, and B. C. d. S. Oliveira. "Compositional Programming". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2021), pp. 1–60.

[120] Y. Zhou, B. C. d. S. Oliveira, and A. Fan. "A Calculus with Recursive Types, Record Concatenation and Subtyping". In: *Asian Symposium on Programming Languages and Systems (APLAS 2022)*. 2022.

[121] Y. Zhou, B. C. d. S. Oliveira, and J. Zhao. "Revisiting iso-recursive subtyping". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–28.

[122] Y. Zhou, J. Zhao, and B. C. d. S. Oliveira. "Revisiting Iso-Recursive Subtyping". In: *ACM Trans. Program. Lang. Syst.* 44.4 (Sept. 2022). ISSN: 0164-0925.